UNIVERSITY OF CALIFORNIA
SANTA CRUZ

HIERARCHICAL STYLE MODELING: A GENERATIVE FRAMEWORK FOR STYLE-CENTRIC GENERATION OF 3D MODELS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Stella Mazeika

December 2019

The Dissertation of Stella Mazeika
is approved:

Professor Jim Whitehead, Chair

Assistant Professor Adam Smith

Professor Luca de Alfaro

Quentin Williams
Acting Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Hierarchical Style Modeling: A generative framework for Style-Centric Generation of 3D Models

by

Stella Mazeika

Style, and the creation of stylized content, is a fundamental concept within Procedural Content Generation (PCG), albeit and under analyzed and under explored one. We see a common thread of procedural systems to designed to create artifacts that look like other, existing ones – from house floor plans evocative of Frank Lloyd Wright's designs [50] to DeepBach, a musical generation system who's stylistic goals are readily apparent [38]. While these systems often accurately capture the style they intend to, they are not without drawbacks – for starters, most of the systems can only encode one style at a time, either as a function of having to train to learn the style, or by manually encoding the rules of the style into the procedure. One avenue that has yet to be explored is separating out style into a first-class feature, either explicitly demarcating the style features or by featuring style as an input to the system.

To facilitate exploring stylistic transformations, I developed a novel framework for encoding generative models, called a *Hierarchical Decomposition*. Based on the philosophical school of *mereology*, this framework emphasizes the part-whole relationships of the artifacts that are encoded. In this way, style is a series of both constraints placed on the Hierarchical Decomposition and transformations that occur to the decomposition.

To facilitate exploring this, I built two systems that utilize this framework to generate vastly different types of content: a smaller system for transforming and stylizing narratives, and a larger system for generating stylized Lego models.

In summary, since style is such a powerful and core feature to procedural generation, creating new frameworks and systems where it is incorporated as a first class feature will allow us to design new procedural systems that allow for a more intuitive design process.

This dissertation is not just the story of my research; it is also the story of my transition – I came to this program, as many do, a very different person than I am today.

Acknowledgments

First and formost, I want to thank my peers, with whom I've spend countless hours in many-sided support circles with, exchanging ideas, excitements and worries. Creating new knowledge, collaboratively, surrounded by so many other brilliant minds, was one of the primary highlights of my time here at UCSC, and in particular, I would like to acknowledge Kate Compton, Johnathan Pagnutti, Sarah Harmon, and Melanie Dickinson for all of the support, mentorship and laughs they've given me over the years.

Secondly, I want to address the most important side project of my graduate career – I learned so much from being a part of Scholar's Play, from both new ideas about individual games, to how to not panic when the audio setup goes awry. And so, I want to thank my core cohort of scholars – Ben Spalding, Johnathan Pagnutti and Stacey Mason – for both putting up with my bad jokes on camera, but for also teaching giving me the chance to learn skills that I continue to put to use to this day.

I want to thank my family — Mom, Dad, and Lizzy. Over the course of this degree, we had a lot of ups and some incredibly deep downs, one that we thought we'd never recover from. However, in the end, the power of love and family kept us all together, and I'm so grateful for all the love and acceptance I've been shown, especially re-introducing myself to you all.

To my friends outside of this academy — Liz Powers, Grace Li, Theo Tiffney, Allegra Bottlik, and so, so many more — thank you for your continued love and support, your willingness to deal with me when I'm going off on wild theories of science, and for being a place I can visit, away from the stress of my ordinary work.

To my mentors, several of whom signed off on this: Jim Whitehead, Adam

Smith, Noah Wardrup-Fruin, and Michael Mateas. Each one of you has helped shape to direction of my research and helped me find my way out of the woods of science more times than I can count, much less list here. I dare say that my time as a PhD student would have been the lesser without the academic support and new insights you all provided.

And finally, I want to thank everyone I've met through the queer corners of the speedrunning and randomizer community that I inhabit. There's more names here than I could ever list, but everyone I've been able to watch, play and talk with has helped me in innumerable ways. From helping me discover and take comfort in myself, to advising me when life threw some curve balls, to celebrating our wins and the joy of finally meeting each other in real life, there's no way I would have gotten to where I am today without these fantastic groups of amazing people.

Chapter 1

Introduction

1.1 Style

> *no, no, no, *dwarves* are art deco; elves are art noveau.*
> — tumblr user eatmystardustloser via outofcontextdnd [1]

Figure 1.1: Examples of an Art Deco door and an Art Nouveau door, taken from [74]

The above quote, taken from an anonymous source on the social media website tumblr, provides a comparison between the fantasy races of Tolkien with modern art movements. On one side, we have Art Nouveau, "characterized by the

predominance of curves, instead of straight lines, and very dynamic, organic and rich decorations, usually stylized, graceful and elegant" [19]. This plays directly into how Tolkien's elves are portrayed (graceful, elegant, nature-loving), and as Destler details in [19], this parallel is one that appears in how their architecture is portrayed. On the other, we have the dwarves, who use feature regular, geometric features in all of their works, to the point where "even their runes and the language itself is as sharp as their architectures" [20]. Unsurprisingly, we see very similar design choices in many Art Deco works, a movement that honed in sharp, geometric features and intricate detail work. In each of these, we see a comparison between the design choices and features of the fantastical architecture and the modern art movements; similar usages of lines and angles, of inspirations and design patterns. As such, we can say that Elves invoke the *style* of Art Nouveau, while Dwarves embody the style of Art Deco.

What we mean by style here is a tricky thing to unpack, since style takes on a number of meanings in both an academic and colloquial context. However, we can start by looking at existing definitions, and building out our understanding from there. As stated in [62] defines style as "a replication of patterning... that results from a series of choices made within some set of constraints." This definition looks at style as part of an artist's process; style is the result of choices, and therefore is the result of specific intent on the part of a creator. In other words, given a number of equally valid choices for creating a series of objects, a creator's style comes from repeatedly choosing the same ones.

In contrast, [29], style is "defined as those distinctive characteristics that enable the observer to link an art work with other works," which is the sense we used above - finding the distinctive characteristics that link Elven architecture with the Art Nouveau movement. This definition doesn't concern itself with the

how or why the works have these distinctive, linked characteristics, and in this way, gives a framework for comparing two unlike artifacts that share features. These similarities can be deliberate or coincidental; the particulars of how these characteristics were chosen is less important than that they exist. Moreover, the definition doesn't provide a particular framework for which characteristics need be considered to link works together — meaning that two pieces could be connected by virtue of having the same artist or by being done in the same historical moment, in addition to specific features of the works themselves.

Beyond the above, this gives us a framework for understanding style in the context of a work evoking a particular aesthetic. For instance, when we look at the Flintstones Car 1.2, we have a car, yes, but beyond that, we have the strong "modern stone-age" style of the series embedded into the car on every front: it's made of wood and stone, and powered by hand (or, well, foot) but it still reads as a car. Other examples of this include choosing a level of fidelity for a 3D model — ranging from the abstract and cartoonish, to the hyper-realistic. By situating a 3D model within that spectrum, the abstraction instantly provides the sorts of links we see between works.

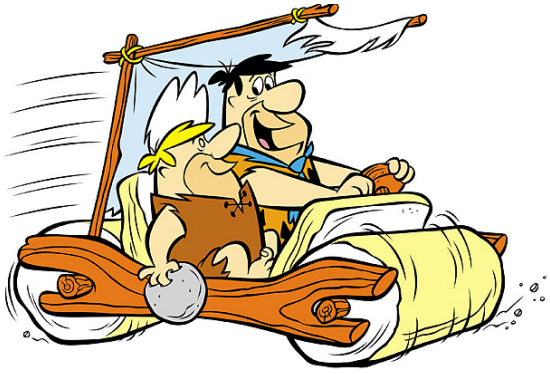

Figure 1.2: The Flintstones Car

From these, we get the picture of style as a term used in distinct contexts, albeit related ones. Style emerges from a series of related artifacts, either via the

creator(s) making repeated choices or merely a replication of traits. As such, style can emerge in any intentionally created artifact, though we often seek to emulate natural objects.

1.1.1 Generative Methods

Make Something That Makes Something

— the ProcJam Website [2]

A generative method is defined as "a function which produces artifacts" [15]. This definition is about as abstract and broad as possible, encompassing everything from computer-aided design systems that utilize complex constraint processes and simulations to ensure that the human designs will function properly when constructed in the real world, to simple systems that generate fantasy names by randomly stringing syllables together. All of these systems operate on a few simple rules:

- They start from input (occasionally, just using some value to seed a random number generator)
- From that input, they undergo some series of processes to generate an artifact

As the examples above imply, the complexity of these underlying systems don't matter for qualifying as a generative method; as long as there is some design goal to the sorts of output from the system, it is a generative method. These design goals often include strong aesthetic visions for the generated outputs of these systems. These range from "I want this game map to be symmetrical" to "I want the system to output songs that sound like the Beatles wrote them" to "I want this Lego car to be Pirate-y." These are fairly straight-forward, as written above,

but they often hide a unexpected level of complexity – it's easy for a computer to understand symmetry, but what does it mean for a song to sound like the Beatles, or for something to look Pirate-y? As humans, we often have the ability to make judgment calls– with some hemming and hawing over edge cases– but for a system to generate an artifact with a particular aesthetic, we need to explain to the system what that aesthetic entails. For the most part, the solution to this problem has been to hard-code

The artifacts a system produces, when taken as a whole, across the entire span of possible outputs from the system, form what is referred to as a *generative space* or *possibility space*. Each artifact in the space is different to varying degrees, and while this sort of showcase focuses on the differences between the artifacts, there are often a number of similarities that tie the different artifacts together — obviously, a name generator isn't going to generate a star-system map, or a 3D model of a tree.[1] Instead, the name generator will likely keep to text output, pulling from a certain set of rules and building blocks to construct the output names. As we explore the possibility space of a generator like this, we start to see repeated elements and patterns in the output — and this are often different than the ones we'd see from a different, but similar generator. Calling back to our definition from [29], it is clear that the artifacts have "distinctive characteristics that enable the observer to link an art work with other works" directly showing that generative systems have their own innate style.

However, I believe that it's important to go a step beyond. While encoding a style directly into a system will ensure that the system will operate within that style, having the flexibility can be more important. Take, for instance, one of the most common uses for Generative Methods — game content. When we talk

[1]Discounting, of course the existence of creatures or cultures with these naming conventions and the desire to generate names for them.

about procedural generation, we often think first of the elaborate systems that power many modern video games. These range from the simple ASCII dungeons of *Nethack* to the elaborate worlds and creatures of *No Man's Sky*. However, as we move towards more and more advanced games, requiring not just more content, but more specialized content, it becomes impractical to build single generators for each specific piece of content.

Consider, for instance, a simple example of a game with three distinct areas, each with vastly different looks and styles to them. One might be a deep, dense jungle, or a frozen tundra, or other similar areas. Each of these needs to be populated with a large number of assets, generated during design so that they look unique. One of these generators might focus on constructing interesting rocks and rock formations for the space — but the needs of a given location are going to be vastly different from location to location. The tree generator for a forest area requires a very different looking space of trees than the one for the tundra area. An assumption, then, might be that to make the different trees for each space, you'd need need a custom generator for each.

However, making custom generators for each kind of content in each distinct space runs you straight into the horrors of combinatorics — at a certain point, these specialized generators stop being 'generators' and start being more akin to tweakable models. Instead, we can gain a lot of mileage by collapsing these generators that provide broader possibility spaces, and focusing on only particular areas of those spaces. But, again, this comes with the cost of needing to tweak each different generator for each different space we want to have it operate in — as we add new spaces, we need to customize each generator to match with that particular space.

I propose, instead, having a single style description for each of the different

areas, that can be packaged up with each of the generators to ensure that that generator in particular operated within the space of the particular style. In this way, we can easily add new generators without worrying about tuning them for each of the styles that we have in our spaces, and additionally, we can create a new style without having to tune each and every unique generator by hand.

1.2 Contributions

In this work, I provide several core contributions, building up a pyramid of specificity from the broadest goal of examining style within a procedural context.

1. An analysis of what style means within a procedural context.
2. A novel framework (Hierarchical Decomposition) designed specifically to promote style-centric artifact modeling and generation.
3. Two systems (Narrative and Solus Forge) that utilize two different implementations of a Hierarchical Decomposition framework to create stylized artifacts in two vastly different domains (fairy tales and Lego models). These provide the following contributions as well:
    - Examples of how to represent artifacts for the given domains
    - A language for modeling and describing specific styles
    - Systems for how to generate stylized versions of the various artifacts.

The rest of this dissertation will proceed by stepping through each of these contributions, devoting a chapter to each, which the Narrative System and Solus Forge each receiving their own chapter.

Chapter 2

Style : A Survey

2.1 Introduction

From color scheme to musical instrumentation to syntax, every commercial or artistic creation can be realized in a multitude of ways. Two different creators can take the same base concept and produce radically different artifacts, and conversely artifacts by the same artist in the same medium can have vast differences in appearance. These differences for the styles of the artifacts. As defined by Meyer, a style is "a replication of patterning... that results from a series of choices made within some set of constraints" [62]. These constraints can vary wildly, from material needs, to preferences of the artist, and can sometimes cause the underlying works to change dramatically.

However, underneath the changes and embellishments of an artist's style, lies some degree of a functional and/or invariant part to the object – a chair, to exist in the class "chair" should provide a place for someone to sit, and because of this intrinsic property all chair necessarily have that feature. These underlying traits, these prototypical elements of objects can be argued to exist separate from the stylistic choices that go into creating particular instances of the object.

Of course, teasing out the differences between the style and structure of any given object is no easy feat – to some extent, this difference fully depends on the class through which we examine the object. For instance, a car's four wheels are usually treated as a structural piece of existing in the class "car"[1] but if we look at the broader category of "vehicle," it is no more of a design choice than a motorcycle's two wheels, or a boat's zero. Ready examples of artifacts being compared this way include songs covered in radically different styles, or films that parody a distinctive filmmaker's style.

Human artists have often treated as having their own distinct styles that they embed into their work. We see this in popular discussions, when people talk about the repeated choices authors make in their works — from Shakespeare's love of wordplay, to comic book artists who have repeated challenges with how they draw hands, to the reductions of various films to the whims of their directors under auteur theory. Because of these repeated choices, we can appeal to our second style definition – style is "defined as those distinctive characteristics that enable the observer to link an art work with other works" [29]. In this way, we can say the characteristics that allow us to link an artist's works together produce that artist's style.

In contrast, the style of generative methods is an underexamined topic. Generative methods are defined as any system that produces some kind of meaningful artifact [15]. Systems exist to generate artifacts in just about every medium from music to game mechanics. It follows that we should be able to discuss the style of a generative method, just like we would with a human artist. According to one definition, generative style is "defined intensionally by a procedure for constructing designs that embody that style" - in other words, by this, the generator itself

[1]Of course, as `https://en.wikipedia.org/wiki/Three-wheeler` shows, 3-wheeled cars exist, but these are decidedly in a class of style decision all to their own.

acts as the definition for the style [91].

This definition, however, only considers style an emergent quality. In contrast, there are a number of generators that intentionally play around with notions of style. These can be divided into two camps - generators that have an explicit style encoded, and generators that take in styles as a parameter to the system. While the latter serves as a perfect example of generative style, the systems that accept style as a parameter seem to subvert it. However, those generators still do have a global style, as there is stylistic information still baked into the system, and the style information passed provides merely a set of constraints or guidelines that the system uses to direct its generation within its style. In other words, one can view the style parameters as sub-styles of the global style.

With these notions in mind, we proceed to examine different generative systems that directly interact with ideas of style and discuss how they both encode and interact the styles contained within. We choose to explore them both as individual systems, but also as members of their class of algorithms, in order to see how the higher level choice of system design can inform the style encoding.

2.2 Grammars

Grammars are a generative technique that take a series of symbols and rules for operating over those symbols, usually taking some set of symbols (commonly one) and replacing them with a new series of symbols. These symbols are divided into two categories – *terminals*, which are the set of symbols that can appear in the final output, and *non-terminals* which form the primary targets for replacement. All grammars include one final feature – a special non-terminal called the *start symbol* which gives the grammar a jumping-off point for its generative process.

The canonical example of a grammar is natural language – in fact, formal

grammars owe their name to the grammars of natural language. In English, for instance, the a start symbol would be **Sentence** [2]. Then, we could have the rules (non-terminals in bold):

- **Sentence** → **First_Subject Predicate**.
- **First_Subject** → **First_Subject** and **Second_Subject**
- **Predicate** → **Verb Prepositional_Phrase**
- **First_Subject** → The **Noun**
- **Second_Subject** → the **Noun**
- **Noun** → apple
- **Noun** → dog
- **Noun** → cat
- **Verb** → was
- **Prepositional_Phrase** → in the house.
- **Prepositional_Phrase** → on the floor.

Grammars, as a general rule, keep evaluating and replacing symbols until the sting is composed entirely of terminals, and so the above grammar will generate sentences like "The cat was on the floor." Here, it's worth noticing that for symbols like **Noun**, there are multiple rules that can activate. Grammars, in general, fall into two broad categories: deterministic and nondeterministic grammars.

[2]Or paragraph, or document, or poem, or any number of other starting points. A sentence is just the smallest fully-contained unit of English

Deterministic grammars will always evaluate to the same output, while nondeterministic grammars have different final states that they can transform into. Within a grammar, nondeterminism comes from points where multiple rules are applicable to the current state. By applying one of those rules, the state may change in ways that either disables or enables various rules, thus making some outputs impossible once the grammar has applied a previous set of rules. If we want to talk about the space of the grammar's domain, it's clear that this is fully defined by its rules [92].

Of course, strings are not the only way that grammars can operate. Shape grammars have been used since the 1970's for style-encoded generation to great success. In the case of shape grammars, the primitives are simple geometric constructs such as points, curves, lines, and polygons [92], though more complex curves are possible. To generate an artifact in a shape grammar, the grammar starts with a seed primitive, also known as the axiom, such as a single point, and then production rules are applied to transform the shape into the output. Production rules are simple, consisting solely of a two components, commonly referred to as the left-hand and right-hand sides of the rule. If the shape of the left-hand side can be identified in the overall object, it can be replaced with the right-hand shape. Often, to avoid the process from continuing indefinitely, a marker system is used. Markers are special labeled points used within the system to constrain both when and where specific rules can be applied to the object.

Most style-focused shape grammar systems focus on highly particular domains, attempting to capture the particular set of design rules that encapsulates that specific space. Common domains include architecture [100] and cars [59], but domains as far and wide as motorcycles and coffeemakers exist in the literature [77, 3]. In particular, in domains as general as 'architecture,' most systems choose

to narrow down even further, seeking to emulate, for instance, the designs of Frank Lloyd Wright [50] or Andrea Palladio [93].

One notable exception to the single-domain focus is the Cross-over Vehicle grammar [71]. In this system, the authors defined the grammar rules for three different classes of vehicle - coupes, SUVs and pickup trucks. Some rules were universal and applied to all three classes, but several were class-specific. To create the cross-over vehicles, the authors used rules from different classes for different aspects of the vehicle. The created vehicles tended to exist in a space beyond simply having feature A from one class and feature B from the second.

Moving away from shape grammars, we see a similar trend in grammar use broadly – since grammars are, at their core, a series of rules that define a domain, it's easy to construct systems that intend to generate within a particular domain. Kate Compton's Tracery [14] is a framework for handing grammars that powers a staggeringly large number of Twitter bots, across a number of domains. For these, the power of the grammar is that the rules give the authors a layer of control and domain specificity.

Approaching grammar styles from a different direction, we have James Ryan's Expressionist [83], a system that uses a layer of tags over the underlying grammar. In this way, the designer can request from Expressionist that the output of the grammar contain some set of tags in the output.

In general, grammars allow for straightforward generation of the output objects in well constrained space, since, in general, the rules that form the grammar will not allow for ill-formed output. This allows us to build rules that create outputs that are not only valid, but are guaranteed to exist within the space of the particular style.

2.3 Multi-agent Frameworks

A *multi-agent framework* is, frankly, self explanatory – a system comprised of multiple independent agents that interact with each other, often representing competing interests. For instance, an example of a multiagent system is the system that Doran and Parberry propose in [23] for terrain generation. In this, there are a series of agents divided into groups based on their spheres of influence: some agents shape coastlines, for instance, while others construct mountains, hills or rivers. The amount of "influence" these different agents have is abstracted into tokens – an agent needs a token in order to create a change in the world. In this way, the user has a certain amount of control over the generative process — by changing the how tokens are distributed between agents, the user can make stylistic changes to the terrain (for instance, giving a lot of tokens to river agents will produce maps with a lot of rivers).

Another multi-agent system, the Letter Spirit project, is described by its creators as an "attempt to model central aspects of human high-level perception and creativity on a computer" [41]. To do so, the creators chose the domain of grid fonts - that is, fonts comprised of letters formed by line segments on a 3-by-7 array of vertices. Letters are represented in the system as collections of decompositions - for instance, an 'x' could be decomposed into two crossed lines, or into two curves that touch at their centers. These sets of decompositions form the conceptual bases that the system uses for the letters, and using these, the system explores in the inherent tension between two forces. The first force pulls each letter towards one of their decompositions, trying to make each letter look as canonical as possible. The second pulls the letters away from that and toward fully embodying the style that the system is using, both towards the preset constraints and the design choices that the system makes while generating the various letters.

The system is made of several interwoven pieces, but at its core, it uses the core architecture developed by the Fluid Analogies Research Group [43]. The system focuses on making small, incremental changes by way of small agents known as codelettes. These allow the system to modify letters, formalize and propagate design choices, and even reverse or modify previous decisions in a fluid manner. This allows the system flexibility in how it generates its artifacts, while ensuring that it never gets stuck by backing itself into a corner. Also included are a set of evaluators that check how close a given glyph is to each letter, and how much the glyph embodies the given design goals. The results of these evaluations feedback into the system and influence what codelettes are active, and what the priorities of said codelettes are within the system.

Techne is a described as a "artbot commune" — specifically, Techne is a framework for constructing artbots that are able to communicate, share their works and learn from each other to expand their concepts of art [72]. Under Techne, bots would have systems for generating art, sharing that art with other bots, and learning new artistic techniques from the other bots in the colony. While this system remained mostly a proof-of-concept, it provided a notion of style embedded into the bots – style, here, being a given bot's preferences for various kinds of 'art' (in the simplified geometric domain that Techne operated in, these preferences exist in the space of "wants a full canvas" to "likes the color red"). As bots communicate and learn from each other, we see the bots learning to better and better make art that lines up with their desires, thus embracing the style that they aspire towards.

In general, multi-agent systems tend to approach style for a number of different directions, in part depending on the role of the different agents, and in part depending on the artifact (or artifacts) that are created – we have everything

from independent agents, with their own styles, creating their own works to agents that lack an independent notion of style working in harmony to design a singular output.

2.4 Genetic Algorithms

Genetic algorithms—also known as GAs—are a class of algorithms that allow for the directed exploration of a complex space. GAs, in short, explore a possibility space by starting with a large population of artifacts that are represented as a series of values. These artifacts, additionally, can be mutated (randomly changing various parts of the artifact) and breed (taking values from two different artifacts and combining them to create a new one). During each iteration of a GA, the highest scoring artifacts (by some metric for evaluating them called the *fitness function*) are taken, mutated and breed to create a new population. In this way, we attempt to find the artifact or artifacts that best fit the criteria specified by the fitness function.

Several style-focused GA systems exist in the literature. Some, such as the systems by Garza and Lores [18], use the GA framework to evolve the outputs directly, while others, such as the system by Lee and Tang [54], evolve shape grammars that then produce the final result. Regardless of the representation of the data, these systems still operate with the same basic loop.

Genetic algorithms are a powerful tool for exploring a large possibility space. However, they do come with their drawbacks. First, GAs require a large enough space that creating a large number of candidate solutions over a multitude of generations is viable. Secondly, the problem must lend itself to being encoded in easily modifiable and mutable ways. And finally, the output must lend itself to being computationally evaluated. While creating a lot of outputs and handing

them off to a human to evaluate can sometimes be easy, this process is very slow and relies on the human's best judgment, which is subjective, even in the best case. In general, GAs need metrics that can be computed from the output in ways that are non-trivial to compute over the representation that gets evolved, however for stylistic features that

2.5 Neural Networks

Neural Networks are a framework for building a trainable computation pipeline, and in recent years, a number of systems have used them to both classify and generate stylized content.

On the older side, we have "What Makes Paris Look like Paris" by Doersch et al. [22] This system, while not necessarily generative, learns to classify the style of a city by examining specific architectural features of that city, looking for elements that form "distinctive characteristics that enable the observer to link an art work with other works" – which, while not mentioned in the Doersch paper, is one of our style definitions [29].

Next, and most famously, we have paper that named a whole genre of system, Neural Style Transfer, by Gatys et al. [31]. In this system, a neural network that was trained to perform image classification is hijacked into forming part of a generative loop where a noise image is mutated to become the fusion of two images. From one image, the algorithm looks for global similarity – capturing the structure of that image: broad-strokes shapes, outlines and forms. The other image's style is used – looking at the local components of the image, and capturing elements like the color palette and stroke quality. The research into this particular area has exploded in the years that have followed, as reflected in [47]. This survey cites over one hundred different papers in this field, each focusing on Neural Style

Transfer through a different lens.

2.6 Comparisons Between Techniques

2.6.1 Style Encoding

The obvious place to start the comparisons is to look at both how the systems encode the style information and what features are encoded as part of that style. Different systems consider everything from color to the exact volume of the artifact, and the considered qualities inform the system's priorities almost as much as the choice of generator.

For grammars, the style is directly embedded into the system by the particular choice of rules. This is readily apparent in systems like the Palladian grammar, since the rules were explicitly constructed to generate artifacts that specifically use the Palladian style. But, even in more general systems, the space defined by the generator still only expresses a portion of the possible space of the domain. If a system only has a rule for using curves in a particular section of the domain, then within that range, there is a style implicitly encoded within the rules - certain sets of shapes and curves are used at the exclusion of others.

There are two major ways that grammars can express style beyond their rules. The first, and most obvious, comes from the choice of rules available at the generator process at the outset. By constraining the set of rules, we can constrain the overall design space to a space smaller than the full expressive range of the artifact class. This is useful if some choices should never appear in the chosen style, or if a particular choice must be made. By redefining the space, we are able to restrict our domain to only the section of the domain that contains the particular stylistic choices we wish include in our outputs.

The other major place that style can emerge from a grammar is the specific selection of choices that are made over the choice of the generation process. If there's a human involved in the choice selection, internal biases come into play. Even if the system is allowed to choose an option completely at random, it is still bound by the global style embedded into the rules. We see this as a core feature of Expressionist, where the output is can be constrained to roll-outs that contain a given set of tags.

In direct contrast, Letter Spirit encodes the style as a parameter; in particular, it handles them as a set of guidelines for the generation process. What makes this system particularly unique, though, is how it handles the generation process given these constraints.

While Letter Spirit lets the user specify certain parameters, it allows affordances for the emergence of stylistic choices from choices made by the system itself. By locking down various design choices during the process, it occasionally over-constrains itself into designing letters that look too similar: for instance, a 'T' without the crossbar looks too similar to an 'L.' Letter Spirit's design allows it to relax the problematic constraints in order to keep moving forward. In our example, one such approach would be to give the T a short crossbar, rather than none at all. This approach allows Letter Spirit to slowly march its way to a stable solution, one that settles nicely in the intersection of both the stylistic requirements and the appearance of the individual letters.

On the other hand, other multi-agent systems take a slightly more passive approach to style, coding the style as a factor of how the agents interact with each other. In the terrain generation system, style is treated extremely hands-off, with the ratios of tokens and agents being the primary control mechanism for the designer, while Techne bots are very much influenced by the social space they

exist within.

Genetic algorithm approaches, for the most part, encode the style that the evaluation functions provide - an output is scored higher if it meets parameters defined as part of the style, and vice-versa. Some of these focus on simple clear-cut metrics - such as the MONICA system, which evaluates its artifacts' styles via simple metrics like "are there between 2 and 10 horizontal lines in the image?" [18]. Other systems, such as Lee and Tang's evolutionary camera system use complex functions of several parameters to evaluate a given camera design's fitness.

In genetic systems that evolve grammars, we necessarily encounter a convergence of the stylistic notions that are embedded with grammars, and the style that is encoded for by the evaluation function. Here, we tend to produce grammars that don't necessarily embody the style, as we use the evaluation function to test how well the grammar embodies the style. These grammars are often deterministic as well, because having multiple outputs would make the evaluation process that much trickier.

Finally, we have the Neural approaches – for the most part, these don't include a formal encoding of what they mean by "style", however their interpretations of style are evident the results that these systems produce. For instance, looking at the features picked out in "What Makes Paris Look like Paris" we see that objects like lamps and street signs have consistent shapes forms within a particular city, and the system identifies this combination of features as the city's style.

For the other learning systems, we see a similar pattern - what they consider to be style is not a formally encoded set of features, but rather the results of the particular learning process. In the Gatys Neural Style Transfer, the 'encoding' of a style is a combination of both the rules that the system learned to transfer style, but also, the style input image itself.

2.6.2 Generation Process

With all the different methods that systems use to encode their different notions of style, two immediate questions arise - how does the encoding method affect the expressivity of the system in general, and how much expressivity is afforded to the system within a particular style. Systems like MONICA or the Palladian grammar offer a large amount of expressive range within the space of their particular style, whereas systems like Letter Spirit allow for a multitude of styles to be expressed, but have a very restrictive notion of how expressive they can be within that style.

Grammars embody a particularly interesting space here, because everything is encoded, either explicitly or implicitly, within the grammar rules. Since deterministic grammars produce identical outputs for each execution, they generally are considered to be a rather uninteresting case. However, it is worth noting that Lee and Tang's evolutionary system for camera grammars outputs deterministic grammars for the camera designs; all of the output grammars use the same rules, just with different parameter values [54].

For non-deterministic grammars, the full expressive range is defined by the complete set of possibilities that the rules can generate. However, for systems like the Cross-over Vehicles system, this can be extremely hard to evaluate. With what the authors call the general modification rules, the grammar allows for its control points to be translated by arbitrary amounts. This means that the grammar can create completely arbitrary output, making automated generation using these rules quite difficult. This is likely one of the reasons why the authors framed their system as a tool for designers as opposed to a generative system in its own right.

The rest of the grammar systems fall in between these two extremes. On one hand, they are still non-deterministic systems, allowing a single grammar to

express a wide range of outputs. On the other, most avoid general modification rules, focusing on a specific set of rules such that they avoid completely arbitrary execution, but still provide a large expressive range.

GAs can afford to have an overly broad domain, since under the generation system, even artifacts that score very poorly under the evaluation criteria can lead to better results down the line. In general, genetic systems try to allow for an expressive a system as possible. Because of this lack of constraints on the initial generation process, the systems are able to cover large amounts of the possibility space before settling in and focusing on a particular segment of the domain. Even when given a particular set of evaluation functions, genetic algorithms can go wildly off the rails, producing seemingly viable generations that evaluate horridly on the criteria used. This can be very useful, as this ability to make nearly unrestrained choices can sometimes allow a genetic algorithm to discover unique solutions in a space of the domain that would otherwise be neglected.

Letter Spirit, in theory, allows for complete expressivity over its highly constricted domain of line segments in a grid of 21 dots. However, it becomes clear that there is a large space within the domain that is never going to be reached by the system. This is not an issue, as the vast majority of glyphs within that space appear as nonsense as opposed to resembling any particular character. Even when we only consider a single letter, there is a large space of possibilities for that character. Where the system starts to carve away parts of the space is when the design choices for one character get rippled through to the other letters in the font. Because Letter Spirit focuses on generating a set of related artifacts, rather than a single one, the design choices that appear in one character necessarily inform the choices that go into others, and the design space for any given character will be highly constrained after a handful of design choices.

The other multiagent systems run into an interesting space of unpredictability because of the nature of how agents interact. In particular, these systems are attempting to model complex procedures as the result of a number of moving parts; as such, there's a gulf of understanding between the parameters of a system and the space of results. However, with this comes a large expressive range even with a fixed parameter set.

As previously mentioned, Doersch's What Makes Paris Look like Paris system does not contain any generative components, however it's easy to imagine a system that takes its results as creates faux districts of a particular city, utilizing the style elements that the system identified to compose that cityscape. On the other, the Neural Style Transfer systems fall into a weird space of having both ultimate expressivity, but also very little: on the one hand, they will operate over any two images handed off to them; on the other, they can't generate anything unless those initial images exist, and do not go very far outside of the those bounds.

2.6.3 Output Evaluation

As a general rule, most systems have a difficult time evaluating the results of their own output. Notions of style and aesthetics are complex concepts even for human evaluators to formalize [68], and this difficulty extends into the domain of generated design. So, most systems rely on some form of human evaluation to judge the quality of the systems output. In some cases, this is a desirable quality. For instance, in the cross-over vehicles, the grammar developed was designed less as a generative system, and more as a tool to guide a designer through the process of designing a car. Even then, the most complex grammars do not have any way of evaluating their own output.

However, there are a few notable exceptions to this lack of evaluation-blindness.

Firstly, GAs need some sort of evaluation metric in order to run their core loop. As they run, they must pass some kind of judgement of all the candidate solutions that they generate. Defining these for a style can be very complex and non-trivial - the simplest is probably demonstrated in MONICA, where Mondrian's style is encoded by hand as the average of eight distinct evaluation metrics [18]. For even more complex styles, such as in [54], human oversight is required to guide the generation process. In that particular system, the authors evaluated the results of every hundred generations and chose particular outputs that were used to inform the next hundred generations.

Even the outputs of the style-identification systems rely on some sort of human evaluation - while they are able to pick out salient features of a style, they have no knowledge of what those sets of important pixels actually represent. The burden of image recognition falls onto the human using the system. While this is sensible (after all, image recognition is a well-known unsolved computer vision problem), it still means that in order for a system to generate an image of a city that looks like Paris, a human would have to hand-label the pictures to use the information extracted by the system. The Neural Style Transfer system comes closer, but only because they choose to emulate the style of how the images are represented, rather than generating new images based on the contents of the images themselves.

One of the only systems that appears to escape having human influences during the generation process is Letter Spirit. By constantly making small evaluations and changes, while keeping a global goal in mind, the system is able to iteratively build toward a solution without any outside influence. However, the trade-off that Letter Spirit pays is the specificity of the domain. In order for Letter Spirit to evaluate the results of its output, it needs to be able to bin the outputs into categories (in the case of Letter Spirit, the letters of the alphabet) to compare

whether or not a given output is too close to a different class than the intended one. Because of this, the Letter Spirit style struggles at developing individual artifacts, providing instead a tool that does an excellent job at creating a coherent array of outputs. The other system that moves away from the human eye is Techne, which is designed from the ground up as a system for agents to interact with and evaluate their works independently of human influence. Here, the agents have a predefined evaluation function (though, the specification does allow for this to change over time).

Chapter 3

Hierarchical Style Modeling

3.1 Introduction

One of the major open problems we wish to address is the issue of stylizing 3D models — given our discussion, we have an idea of what style and stylization means, but there are a number of concerns that apply uniquely to 3D models that have this a challenging problem. Firstly, when we wish to stylize an artifact, the changes we wish to make are context-sensitive (when styling a model of a car, we might wish to make changes to the main body that we don't want to carry over to the wheels). This implies that some degree of semantic segmentation of the model itself.

Secondly, sometimes the sorts of changes we want to make to the model run deep – adding or removing entire subsections of the model to fit within a style. Examples include, for instance, making something look more "pirate-y" by including a treasure chest, or by adding a pirate flag on top of the model. These changes go beyond simple re-texturing; the entire mesh itself needs to be transformed to align with these new (or removed) parts. And while having the segmentation makes this simpler, understanding how to take and rebuild the mesh with new

parts remains to be seen.

One way to handle this problem is to represent the model as a collection of parts that are pieced together via a series of positional constraints. Then, assuming that the piece-combination is well-formed, it's simply a matter of performing a 3D constraint solving task to place all of the pieces without violating any of the constraints. However, constraint solving in general is a very heavy-weight and powerful tool, and so if we only have a single model we wish to represent, this method is overkill at-best — but if the model is underconstrained, then we can can have multiple possible renderings represented by that particular model. So, if we start viewing these less as singular models, and more as model-spaces utilizing the power of constraint programming gains value.

We can further justify this use by taking the generative nature of these models a step further – if we have individual parts that we want to have generative elements, beyond just their positioning, we need some way to ensure that the generative elements play nicely with each other, which is a problem that, in the general case, requires the power of constraint systems.

In this chapter, we propose the *Hierarchical Style Modeling* framework for modeling both artifacts and generative spaces. This framework was specifically designed to allow for the modeled artifacts to be easily transformed using style rules, and takes a lot of its inspiration from how Cascading Style Sheets (CSS) interact with different forms of HTML. In particular, the Hierarchical Style Modeling forms the HTML portion of the pair – a structured model of the core object that can be transformed in a number of ways using the external style information.

This framework was designed with the following goals in mind:

- The model should describe both singular artifacts as well as generative spaces.

- The model should be constructed with different related portions of the model grouped in logical units
- The model should be designed such that it can be transformed with ease, both in terms of bulk transformations and individual transformations

This framework is designed specifically to address the considerations of transforming 3D model geometry, in ways designed to handle generative components. In the rest of this chapter, we will look at other, similar frameworks and theories before doing a deep dive into the specification of the framework itself.

3.2 Other Frameworks

While mereology provides the primary influence for this work's structure, this is by no means the only framework for organizing data into a tree structure. What it does provide is a particular understanding of how the layers of the tree are related to each other. Here, we briefly examine similar frameworks with an eye towards how they structure, and allow for the transformation of, the data they contain.

3.2.1 Containment Modeling

Containment modeling, as defined by the authors who proposed the model, is a "specialized-form of entity-relationship model in which the only allowed form of relationship is a 'contains' relationship" [35]. As an entity-relationship model, the two primatives of the model are the aforenamed *entities* and *relationships*, Entities come in two major categories: containers (which can contain other entities) and atomic objects. There are, additionally two major types of containment: inclusive

(where the actual object is contained within the container) and referential (where the contained object is stored elsewhere but can be referred to).

These abstractions are extremely important for talking about a system that seeks to model how digital files and data are stored and organized relative to each other: does a Java class contain this other class; and if so, does destroying the container necessarily destroy the contained object? The particular abstraction of referential containment makes less sense when talking about physical objects, outside of contrived examples involving (for instance) a backpack full of books, versus a backpack containing the library Call Numbers of those books. However, the framework itself offers a very detailed and precise framework for describing the relationships and the requirements of each of the different containers, such as how many and what kinds of things can be stored or the ordering methods imposed on the contained objects.

However, the one abstraction missing from this is an understanding is a notion of physical position and relation between the objects. Since Containment Modeling is a framework designed with the digital in mind, it lends itself more naturally to exploring virtual relationships.

3.2.2 UML

UML, the Unified Modeling Language, is a language for writing software blueprints, providing tools to "visualize, specify, construct, and document" the various portions of a software system [10]. Arguably, the Hierarchical Style Modeling framework is a software model (especially when one considers the generative nature of the system), and one could use UML to model the structure instead. However, UML diagrams come with a lot of extra baggage that are at once both useful and harder to read that the more simpler notation we will describe below.

In particular, this will come into play once we start creating more interconnected and nested diagrams, where taking the time to add an addition box onto each edge would dominate the space of the diagram. This is effectively the argument made in [35], where the authors justify their (much simpler) diagrams for containment modeling.

Despite this, UML has an additional construct that we don't address here. Since UML is specifically designed in model digital objects, it includes a notion of two separate containment relationships – aggregation and composition. In composition, the contained items are directly "owned" by their container, and changes that affect the container will extend down (the common metaphor is that of a backpack that contains some books – if you leave it out in the rain, the backpack and books both will get drenched). In contrast, you have the aggregation relationships – in which, instead of having the objects themselves, the container merely has pointers to those objects instead. This allows for the container to undergo a number of changes (such as the rainstorm above) while the contained objects remain unaffected. This difference is critical in computer science, where tracking whether or not a container "owns" a particular object determines how to handle things like garbage collection. However, when looking at more concrete artifacts, this sort of dichotomy doesn't quite provide any use – since the parts of the object are physically present, the separation provided by aggregation doesn't quite work in this context.

3.2.3 Constructive Solid Geometry

Constructive Solid Geometry (or CSG) is another technique for constructing solid models, by taking an initial set of 3D primatives (typically, simple objects such as spheres, cube and cylinders) and conbines them, using simple Booelan

set operations (union, intersection, difference) as well as geometric transformations (translation, rotation, scaling) [79]. This technique is fairly similar to the Hierarchical Style Modeling that we propose below, with a few notable exceptions — first, the way that the CSG model constructs models from individual pieces does not constain any notions of semantics; it purely focuses on the geometric conponents. This sort of construct operates on a much lower layer than we wish to for the HD model to operate, as while we could model individual semantic units of the model as CSG constructions, we would need to add in some kind of framework on an abstraction layer above the CSG model to handle the semantic segmentation and the management of various sub-parts.

In short, CSG would make a reasonable framework for modeling the underlying geometry and the generation of said geometry, but on its own, it would be insufficient for our purposes.

3.3 Mereology

Mereology is the philosophical and mathematical study of parts and wholes, forming a theoretic backbone for the Hierarchical Style Modeling framework. Since our intent is to model objects as combinations of other parts, I start by a quick walk through the theory and some basic definitions.

To begin, the most basic mereological concept is as follows: all things in the universe are either atomic objects, or they are composed of two or more *proper parts* (defined as any part of an object not equal to the whole). Expands of proper parts abound in the world, from the trunk of a tree to the canvas of a painting, but what's harder is finding atomic objects (also known as atoms), or objects that have no proper parts. As Simons remarks in [84], not even our real world atoms are truly atomic — they're made of protons and neutrons and electrons, which are

themselves made of even more particles. Because of this, there's some amount of contention among philosophers as to whether or not atoms exist in the real world, or if we need to consider building this theoretic model to not include atoms. For my purposes, atoms are useful, as they provide a computational system a way to ground out the models, avoiding an infinitely deep nesting of parts.

Parthood is transitive (since if a part X of a part Y of an object Z, then clearly X is a part of Z); however it is neither reflexive (X is not a proper part of itself) or symmetric (X being a part of Y does not imply Y is a part of X). From this, we get that parthood gives us a partial order of components of an object — for our purposes, we'll assume that this forms a tree, however, the theory does permit distinct parts of an object that share subparts. This forms the basic layer of mereological theory.

From this, [84] continues to add new layers to the previous theory, considering things such as "temporal parts" (such as the start versus the end of a race) or how to model "essential parts" (things that are required for a particular object to exist — ie, all people have heads). However, there is one important characteristic that Simons does not explore — that of the physical relationship between an object's parts. A car might necessarily require an engine, but in order to run, the engine needs to be placed and connected to other parts is very particular ways. Under our previous mereological framework, there's no difference between a collection of car parts and the fully assembled car.

To account for this, Koslicki introduces the notion that real-world objects fall under the framework of *structured wholes* — that is, the physical arrangement of the object's parts is itself a part of the object [51]. Consider a water molecule — two hydrogen and one oxygen arranged and connected in a very particular way. Under this framework, we can say that molecule has 4 parts (at minimum):

the two hydrogens, the oxygen, and their configuration, which includes both the angles at which the atoms are arranged, but also the nature of the covalent bond.

3.4 Hierarchical Decomposition

A Hierarchical Decomposition is an abstract structure, built to structure the parts of an object or generative space for ease of transformation and generation. To do so, we organize the parts of the object into a tree structure, where each node of the tree represents a part of the object, and a node's children are its proper parts. In order to reason over a model like this, we need to have atoms that form the base of the model. What the atomic units are is very domain specific, but for a physical object, these are the smallest parts that we want to reason over — the legs of a table can be treated as their own unique parts, or the windshield of a car.

Figure 3.1: A lamp, found at `https://commons.wikimedia.org/wiki/File:Tiffany_dragonfly_lamp_with_pigeon_sculptures.jpg`

Let's concretize this with a particular example: a lamp, namely, the lamp in Figure 3.1. We can view this lamp as being made of a few parts: the metal base, the stained-glass cover, the light bulb itself, and then the electrical fixings

that connect the bulb to its power source. These atomic parts could be unpacked further, in theory — for instance, treating each pane of the stained glass as its own atom, or viewing the filament of the lightbulb as a part distinct from the glass surrounding it. These further decompositions have their uses — if we care a lot about how the stained glass is modeled, constructed or modified, we might wish to focus in on the details of that part. This is a choice that we can make in our modeling procedure, so for now, we say that there are only the four parts initially listed.

With our atoms, of course, comes structure. There are a number of ways we can organize four unique elements into the tree, but we want to pick a structure that organizes the parts of the lamp into chunks that are useful for our purposes. Pictured in Figure 3.2 are three different ways we could place our atoms into the tree. Of note are the additional "Attach" and "Core" nodes — these nodes merely serve to provide identifiers for the part of lamp comprised of their children — in the upper left tree, the "Core" is the electrical wiring and the base, representing the non-removable parts of the lamp, while the upper right tree includes those parts in its "Core." Both of these are equally valid, and communicate merely a difference in perspective on how the lamp should be organized; one could imagine another version of the tree that features the "Core" node splits into the bulb and a "Stand" node (which is equivalent to the the UL tree's "Core"). Going forward, we'll work with the Upper Right tree as our framework for the how the model is structured.

While all three of these decompositions are valid on paper, they have differences that make the choice of one over another relevant, in terms of usage. One of the features of a well-structured Hierarchical Decomposition is that nodes are collapsible – ie, a non-terminal can be compressed into a terminal by "tucking"

their children up. While this seems trivial, it will become more relevant as our understanding of these decompositions expands.

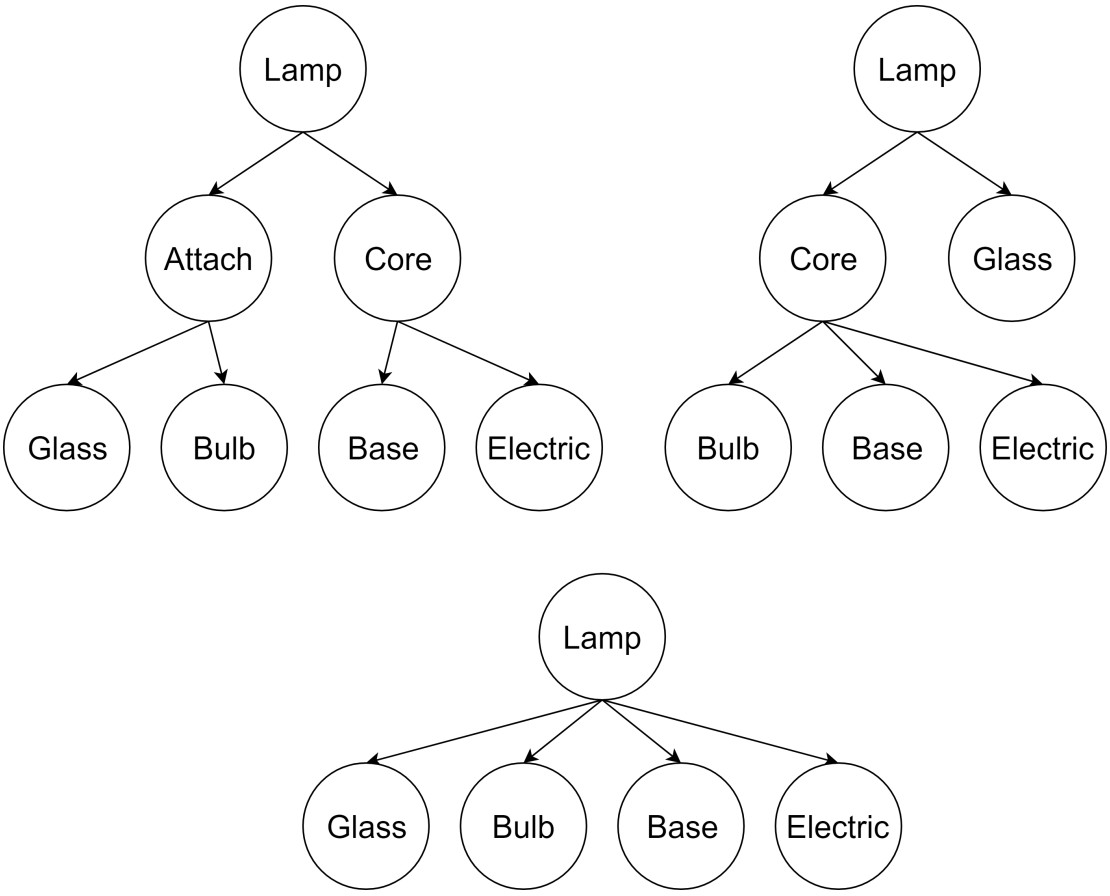

Figure 3.2: A series of three trees, each organizing the different atoms of our lamp in distinct ways.

With that, we have an organizational map of the object already in place — the tree structure conveniently divides the various parts of objects into logical groups. However, an extra step needs to be in place, as right now, we don't have any structural information in the model, just the parthood relations. Harkening back to [51], we want to include structural information so we can use the Hierarchical Decomposition as a map for recreating the object encoded within it. To do so, we include an extra series of edges that define relationships between pairs of parts. These relationships, called *structural constraints* enable us to define how the object

is reconstructed from the decomposition. Depending on the modeled object, these can take on a number of different appearances, ranging from geometric positional constraints (the wheels of a car go under the body), to identities (i.e., two parts need to be made in identical fashions). These form a graph, separate from the tree structure, that links all of the parts together, so that each part is has a connection to some other part of the object.

Going back to our example, we'll want to start by considering the relationships between the different parts of the lamp. Let's start with some of the obvious ones—the electrical components of the lamp is passes through the center of the base, and the bulb screws into the top of that piece. Finally, the glass portion rests on top of the base. Bringing this into our diagram looks something like Figure 3.3. Notice that we're just using natural language descriptions of the relationships—when we're encoding this for a computer, we'll want to provide a detailed specification of what these individual relationships mean. However, which constraints we implement and their specific implementations are left to the individual models and systems, because of how context and domain specific they tend to be. There are often some general ones we want to include (part A is on top of part B as an example), but we will provide a full description of specific constraints in later chapters.

Of course, these constraints can often be under-specified, signaling that there are multiple valid configurations that the model can take on—it doesn't matter exactly where we put the legs of a table so long as they're all mostly at the corners, for instance. In our lamp example, we have a few spaces were we can play around with how the pieces are put together, such as how deeply the electrical pieces are set into the base of the lamp. This sort of flexibility allows us to use the model to describe a generative possibility space, but we can take that even further.

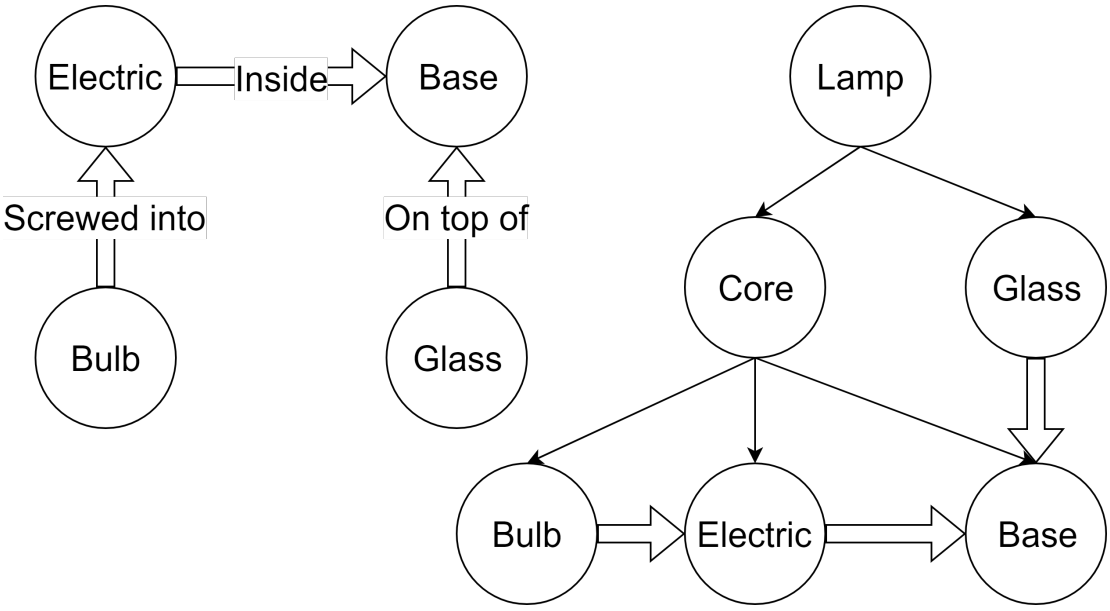

Figure 3.3: The positional constraints between different parts of the lamp

3.4.1 Generativity in Hierarchical Style Modeling

Beyond having the flexibility in positioning that can come from loose constraints, we can introduce more generativity into our model by allowing the atomic parts not to be concrete instants of objects, but allowing them to exist as generative processes that are resolved at the same time as the model in total, and are used to create an instance of the atom.

That's a lot of word soup, so let's look at how it would play out in our example. Here, we're moving away from a model that encodes the particular lamp above, but instead represents a set of lamps that follow the same basic structure as the lamp. One place where we could have a generative chunk is the stained glass — the lamp we have pictured features dragonflies, but it's easy to imagine a number of possible lampshades that could be put on the lamp instead. We could, then, instead model the Glass node as a node that picks a random shade when realizing the model.

However, not all of generative nodes are this simple to resolve. Sometimes, the way a particular node is resolved can come into conflict with the constraints between that atom and other atoms, and if there are multiple generative atoms in the model, the way that those other atoms are resolved can be impacted. For instance, if the base of the lamp is generative (similar to how the glass was), this has an impact on the length of the wiring — if the wiring isn't long enough, it won't fit properly into an extremely tall lamp base.

Going another step deeper, we can expand on the notion of the generative nodes, which merely allow for individual atoms to be generative elements. We can also have non-deterministic non-terminal nodes: internal nodes that allow different sub-trees to be chosen between when resolving the decomposition. This allows us to increase the flexibility of our representation, giving us the ability to represent a diverse set of structures within a single decomposition.

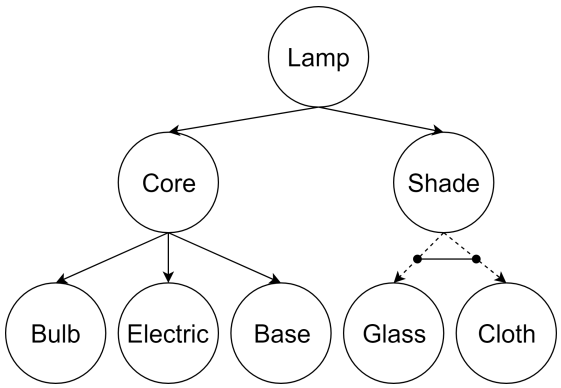

Figure 3.4: An example of a non-deterministic node, in the lamp model — providing a choice between a glass shade and a cloth shade.

Going back to our lamp example, Figure 3.4 shows a way of representing the lamp's shade as a non-deterministic entry, a choice between a cloth shade and the glass one we've seen thus far. None of the constraints need to be changed for this, as all of the constraints that would apply to either shade are captured by the constraint graph back in Figure 3.3.

The non-deterministic nodes pose an interesting challenge for the constraint graph, additionally: namely, how do we ensure that a constraint between an atom outside of the ND node applies to an atom within the ND node. Here, we choose to, instead, apply the constraints to the node itself, and then, upon resolving the non-deterministic node, we pick an atom within the node to apply the constraint to.

3.5 Transformation and Stylization

With all the previous framework in place, we have a generative model described by the Hierarchical Decomposition. This is a powerful tool for describing a single space of an object, but our goal here is to stylize the models described under the framework. So, how do we go about doing that?

We begin by describing the style as a series of rules, or constraints, with the understanding that any model that satisfies these constraints is considered to be stylized appropriately. These constraints can come in any number of forms, but there are a few common patterns:

- A specific atom or subtree needs to be included in the model
- No atoms of a given type are allowed to be included in the model
- All atoms of type X are replaced with a specifc atom or subtree
- All atoms of type X must have the property Y.
- Certain pairs of atoms must feature a particular relationship to each other (such as positional constraints, or be composed of identical material, etc.)

These constraints allow us powerful tools to describe styles in terms of the atoms that are or aren't included in the model, how those atoms are realized, and

how those atoms relate to each other. In order to make sure that the system can interact with the atoms correctly, we need one extra layer in the system — we tag all of the atoms with metadata describing the stylistic properties that they feature. This way, we can be sure to only apply the style rules to the proper pieces of the model.

Let's go back to our lamp, and explore applying a style to it. First, we need to pick the style we wish to apply, as well as specifying the particular rules that go into that style. To start, let's work off of the decomposition in Figure 3.3, and consider the following style rules:

- All glass components of the model are replaced with cloth versions instead.
- The electric components are outside of the lamp as opposed to inside.
- A (tasteful) handle is attached to the side of the base.
- The bulb is removed from the model.

While this won't necessarily leave us with a functional lamp, it does illustrate several of the major ways that a model can be transformed.

We address this modifications one at a time; first, we can simply replace the lamp's glass shade with a cloth version. Handling the second rule is trickier, since the style rule technically contradicts the model. To handle this, we bow to the style rules over what the model specifies — by allowing the style rules to take precedence, we ensure that any applicable rule gets expressed in the new model.

To handle adding in a new part of the model, we need two things: a place for it within the hierarchy and a positional constraint that ties it to the rest of the model. Placing the new part within the hierarchy is straight forward if we already have a constraint to attach it to another part – we can just couple them

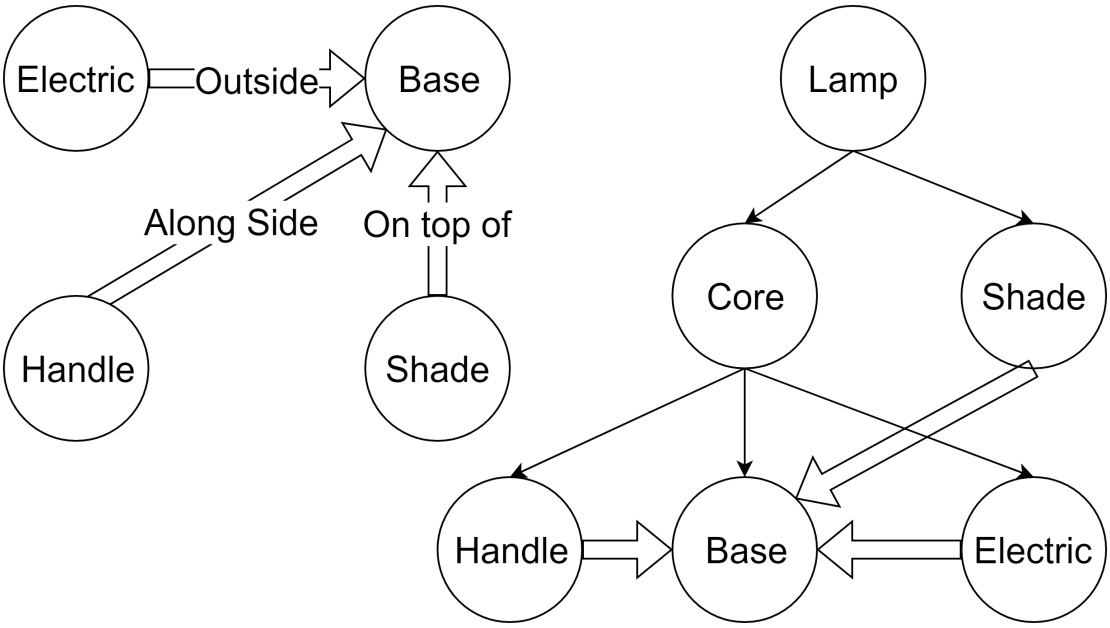

Figure 3.5: The lamp diagram from Figure 3.3, after the style transformations have been applied to it.

together. Deciding on what constraints to include is a much thornier problem, so instead, the specification calls for including a new constraint, and a class of part that the can form the other side of the constraint. Since these styles are meant to be generic and independent of any given model, we don't want to lock the style into requiring a particular part[1].

Finally, we remove the bulb from the model. Notice that this (as Figure 3.5 shows) necessarily removes the constraints between the bulb and the rest of the model. Since all of the parts that we are removing are leaves, this does not cause any issues for the decomposition – we can just remove the parts without concern. However, where this gets us into trouble is the constraint graph. If the part being removed leaves the constraint graph disconnected, we have an issue – we now have two Or more independent objects instead of one continuous one. So, if a part being removed creates this gap in the graph, we create new constraints

[1]Of course, if the target has no applicable place for an addition to happen, then we skip it. But it's not for a want of trying

between the now-disjoint pieces of the graph, picking parts from either side of the graph to form the edges, until the graph is fully connected again, using one of the constraints that the two parts shared with the now removed part.

We can see the final results of this transformation in Figure 3.5.

3.6 Examples

Now that we have our framework, it's important to consider that frameworks can be used in unusual ways and cases, beyond just the initial space considered. Here, we present a few different examples of Hierarchical Style Models in a few different domains, beyond the 3D object case we've utilized thus far.

3.6.1 HTML

HTML, the markup language that powers the modern internet, is a quintessential example of how to structure a hierarchical space. In HTML, angle brackets define markers, providing a space in which text is rendered to the browser differently than without. Additionally, these markers can be nested, providing everything from applying multiple text features (such text being a hyperlink as well as bold) to completely changing the context what's written (script tags call out Javascript code, and title tags demark the name of the document). Additionally, information can be placed in tags that describe blocks to control how they're laid out — block elements such as divs control the placement of their contents, and the requirements of a given div can interact with others, as well as the resolution of the web browser, in determining where all of the divs are placed.

By default, HTML describes a static document, given a particular browser resolution — it should always render the same way when its loaded. However,

we can inject some interactivity and variation with the other two key pillars of website creation — Cascading Style Sheets (CSS) and javascript. Precisely as our HD framework describes, CSS files are external data that describe stylistic features that are applied to particular aspects of the HTML document. Of course, sometimes these features can lead to contradictions — one rule says that a specific piece of text should be colored blue, another says that all text should be red. This is where the cascading part of CSS comes into play – it includes a built in mechanic for deciding which rules take precidence by looking at the rule's specificity — more general rules are overwritten by more specific ones, allowing designers to create style sheets with a base background set of rules that form a baseline style, while further in, the more specific rules take care of their particular cases.

Below, we have a sample HTML document structure (with some details removed for simplicity). In Figure 3.7, we have the graph version of this particular layout; the nested structure of the HTML document translates directly to the hierarchical style graph.

```
<HTML>
   <head>
      <title> ... </title>
      <meta> ... </meta>
   </head>
   <body>
      <div> ... </div>
      <div> ... </div>
   </body>
</HTML>
```

Figure 3.6: An example of an HTML document

As described above, the stylization and transformation process is simply: We can easily add additional divs and other HTML components to a document, remove others, etc, as well as changing how the document is rendered via applying

CSS files to the document.

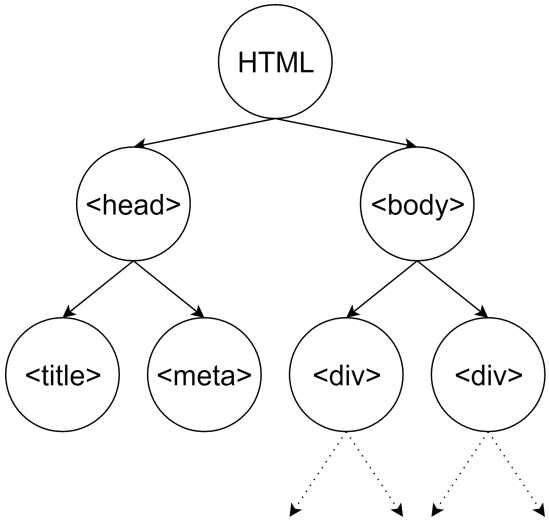

Figure 3.7: The Hierarchical Style Model of the HTML document in 3.6

3.6.2 Spelunky Maps

Spelunky [102] is a roguelike platformer game, in which the player explores a series of procedurally generated levels in order to find the exit, as well as treasures and exploration tools along the way. While each level is generated each game, there are some consistencies across the various levels. First, and most obvious to players, is that each chunk of four levels is themed - first, we have a sort of classic dark-brown-rock cave, but this makes way for an underground jungle, a frozen ice cave, and others. In addition to each world sharing a sprite set, each has some different mechanical components as well - different enemies, different obstacles and different special stage elements appear in each of the worlds, though the level in which they appear can be random.

This provides a useful framework for understanding the style of Spelunky levels — each one of the worlds has its only style, which we can apply to a 'default' Spelunky level generator to create the levels for the various different worlds. But,

this begs the question – does this platonic ideal of a level generator exist, or are the different worlds generated via completely separate and disparate means?

Luckily for the argument I'm constructing here, there is such a generator. Darius Kazemi provides a detailed explanation on his website [49], so a summary will suffice here.

Each Spelunky map is comprised of a four by four grid of rooms, each of which has a fixed size. At run time, the generator picks a room in the top row to be the starting room, and then randomly moves left, right or down — until it tries to move down from the bottom row. That bottom room is marked as the exit to the map. All of the rooms in this sequence are called the "critical path" – the intended route through the level, and each room off the critical path is labeled as a side path. With the critical path in play, the game is able to ensure that all of the rooms are connected – leaving room for the player to transverse the level without the use of excess resources. From there, each room has a number of templates that it can have, based on the different exits that the room has (ensuring that there's always a path through the level that doesn't require the player to destroy a wall. Each of these templates, then, can be resolved in a number of ways, with a number of optional tiles, enemies and treasures that can be added in. In this way, each map is generated from a large possibility space of similar looking maps, but with a critical path directly incorporated into the design of the map.

In Figure 3.8, we can see the hierarchical style model of the Spelunky map, with each of the rooms treated as the atomic units [2]. Notice that all of the rooms have connections to all of the adjacent rooms, which are used to form the

[2]Here, as with most places, there is an argument to be made about what we should treat as the atoms — an argument can be made for going down to the tile level, with the tiles being component parts of the rooms. However, this is more complex overall — requiring a multitude of extra nodes — and gains very little value, as the room layouts are chosen independently of the other rooms, as well as dependently on each other. So, by instead treating each room as an atom, we can have each of them as a separate unit

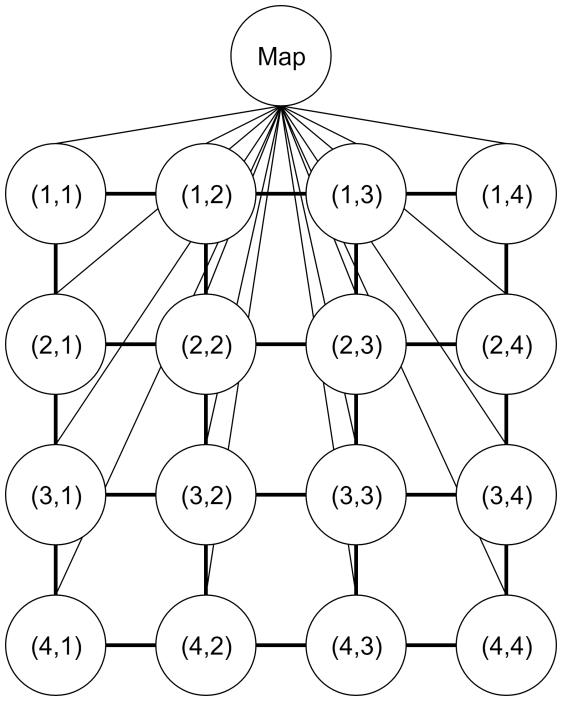

Figure 3.8: An example Hierarchical Style Model of the levels from the game *Spelunky*

traversability constraints, based on the room types. Each of the room generators is identical, and generates a room as described above.

In our model, we can then treat the description of the critical path, as well as the world (which, as mentioned above, define the tile set and room layouts that are used in the generation process) specification, as the style for the Spelunky level, controlling generation and adding in additional constraints onto which specific layouts can be chosen for each room.

One will immediately note that the critical path through a level is not a style in a traditional sense, but fulfills the roles of a style perfectly — providing additional constraints that change the form of how the different atoms are realized.

3.7 Conclusion

Hierarchical Style Modeling is a powerful framework, precisely because it expands on the tree structure that underlies many of the core data structures in computer science, and expands it with a structured framework for understanding how to transform the different parts that tree to meet particular design requirements.

Chapter 4

Story Transformation

4.1 Introduction

Narrative transformation is the act of modifying a pre-existing narrative, whether to serve some purpose or to fit an externally imposed constraint. We see a ready example of this in adaptations, where an existing narrative is taken and transformed to fit the conventions, strengths and limitations of a new medium. When books are adapted into films, for instance, characters are combined or removed and plot lines are dropped to allow the narrative to fit within the appropriate length of time. Additionally, when telling a story, we often change details to fit our audience. For instance, a parent will skip events deemed inappropriate for their children when reading a book, or the creators of a film adaptation of an old cartoon will make a new narrative to fit what they believe the now-older audience would like.

In this chapter, we take a quick detour from examining style through the lens of visual artifacts, and look at a radically different medium, with radically different tools and frameworks for operating. Here, we propose a system designed to make targeted modifications to a semantic representation of a narrative, de-

signed around changing the narrative to fit some higher level goal. We provide a rules-based system for describing the constraints and transformations that will be imposed on the narrative to fit the given requirements. This work serves as a primative approach to the Hierarchical Decompostiiton framework expanded upon in the previous chapter, and utilizes a very different logic programming framework than that seen in the next system. However, it represents a milestone work in the process of developing the further theories.

4.2 Related Work

Several different techniques have been commonly used for narrative generation. In [44], the authors provide four separate methods for generalizing Propp's theory of narrative grammars. The initial grammar that the authors describe operates over 5 levels of increasing specificity – the system first determines the high-level structure of the narrative, and at the bottom level, determines the individual actions that the characters take.

Additionally, we have systems such as Minstrel [99] and an improvement, Skald [95], which generate novel narratives. Minstrel operates by modeling some aspects of creativity to generate novel stories. To do so, the authors rely on transform-recall-adapt methods, or TRAMs, heuristics that offer solutions to problems that may arise during the generation process. In this way, it is able to transform the representations of narratives to fit an encountered narrative model. Using this technique, dubbed imaginative recall by Turner, Minstrel is able to generate novel narratives from an existing repository of reference narratives. Skald does not differ from Minstrel on the surface, but instead serves as a more-modern re-implementation that codifies and strengthens Turner's work by improving the TRAM selection process and updating several other mechanisms within Minstrel.

Finally, we have two systems that construct new stories from existing ones by analogy: the Story Translator [80], and SAM [70]. Both systems operate using very similar approaches, to the point of the authors in [70] observing that the Story Translator can be modeled within SAM. Both systems focus on taking existing narratives and converting them into an analogous one in a new domain; one example given is a fantasy story of a kidnapped princess being reworked to invoked cowboys holding each other at gunpoint. In contrast to the system described here, the authors view the changes in the events as a side-effect of the systems not being able to find a perfect mapping between the domains. However, these systems do provide an example of modifying a narrative to fit a given set of constraints.

4.3 System Description

Within our system, we have three representational elements that we need to consider: first, our representation of the narrative itself; second, the ontology that the rules operate within; and third, the set of rules that define the restrictions and transformations that we wish to impose upon the narrative.

To encode the stories that we wish to transform, we use a novel semantic framework, Rensa [39], to encode the representation of the narrative itself. Rensa is designed to be a generalization of Elson's Story Intention Graph [25], and as such offers all of the affordances that SIGs do while also incorporating a semantic network representation of the space around the narrative. In this framework, we represent characters as collections of mutable attributes, ranging from character tropes (protagonist, villain, etc.) to emotional attributes (happy, afraid, etc.). Because of the flexibility of Rensa, he particular set of attributes that comprise a character is defined by the user, and the set we choose is elaborated on in Section

5. The narrative itself is represented as a series of actions taken by the characters.

On top of the description of the narrative, we also provide the system with an ontology of the domain that the story takes place in, specifically, information about particular actions or character types. Similarly to other works that use similar ontologies to generate stories [69][80][70], by defining this representation of the domain, we gain two major benefits. First, it allows us to reason over properties and actions that are not part of the initial narrative. If we want to substitute an action for a less violent one, for instance, we need a set of possible actions to choose from. By having and maintaining an action ontology, we are able to ensure that the action we choose fits within the setting for the narrative. Secondly, this ontology allows us to specify additional information about the encoded elements. For example, in order to remove violent actions from a narrative, we need some notion of how violent an action is. By including this non-narrative-specific information, we gain the ability to reason over the additional information that we have included in our ontology.

This, of course, is radically different than the hierarchical decomposition model on a number of fronts – ranging from the lack of overall hierarchy[1] to a lack of positional constraints between the different parts of the model. While it would be possible to shoehorn in some of these, and bend data to fit the Hierarchical Decomposition framework, utilizing a framework designed to fit the objects in question is the better choice in most cases.

[1] it's easy enough to group event sequences as existing as the children of an event node, but how do we explain the relationships between the characrters, or location? Having high-level concepts as internal nodes makes sense, possibly.

4.4 Rules Representation

With the narrative and the space of the narrative we wish to generate in, we now need to build our series of rules for transforming the narrative. To do this, we take our semantic representation and encode it in a Rete algorithm-based rules system [30], which allows us to efficiently identify and execute rules over a collection of statements. The state is represented as a collection of facts, which in our case are simply the assertions that we have created to represent our narrative and the information contained within the ontology.

Rules can have three different effects when they fire: they can add any number of new facts to the system, they can modify any number of existing facts, or they can remove existing facts. When the state is changed in this way, we have a new space of rules that will be able to fire next.

In this system, we implement several major classes of rules that represent different kinds of transformations that we want to be able to make to the narrative representation. Note that not all of these can be represented as a single rule, but sometimes instead must be implemented as a set or series of interrelated rules to produce the effect desired. Side effects are certainly possible, but should be minimized for rules that may need to fire multiple times.

First, we have a collection of general types of rules that we use within the system. These include rules to include additional information or actions as required (for instance, some narratives will require all of their characters to have names), to modify or remove existing components of the narrative, and, most importantly, to ensure consistency within the narrative because of these modifications. As the rules fire, we may find ourselves in states that are not valid – perhaps an action is added to the narrative, but the character performing the action does not exist at that location at that time. The consistency rules fire to trigger modifications

to ensure that these issues are resolved.

With the general sorts of rules in place, we then incorporate a rule set that defines the requirements for the target narrative. Beyond rules that can be modeled in the above class (such as the removal of violent actions), we can include rules such as requiring a character of a particular archetype to be present (whether that character has to be added to the narrative on the fly, or an existing character can be modified to fit the archetype).

Once we have our rules set in place, constructing the modified narrative happens by allowing the Rete algorithm to use the rules to modify the narrative until it reaches a state in which no further rules can fire. At that point, we take the current state of the engine and transform it back into the semantic representation, so that we can examine the output.

4.5 Example

To test this framework, we focused our efforts on constructing an adaptation the story of Sleeping Beauty, since there's a culturally salient transformation for that particular story – namely the Disney adaptation of the original story from the Brothers Grimm. To do this, we used the text provided at `http://www.pitt.edu/~dash/grimm050.html` as our reference for the original tale [6].

First, we discuss how we have encoded the narrative into the system. Characters are initialized with the following attributes: Name, Gender, the character's Narrative Role (Protagonist, Villain, Sidekick, etc.), and any other necessary attributes (Character is a faerie or an animal, etc.)

Then, the narrative is comprised of three different kinds of statements: a character can take an action, a character can move to a new location, a character can acquire a prop, or a character can gain or lose a property. Each of these has

the following attributes:

- Actor (The focus character)
- Type (Action, Location, Prop, Property)
- Subject (The Action taken, the Location moved to, the Prop acquired, or the property gained or lost)
- Time Step(s) (This statements can have one of two time step attributes – either the statement happens at a single time step, or it has a time step where it starts and a time step where it finishes)

In our encoding of Sleeping Beauty, the narrative happens over 12 time steps, consisting of 26 different statements.

The system uses the following rules to create a mapping from Grimm to Disney:

- Every character has a name.
- The hero has an animal companion as a sidekick.
- Repeated characters, or characters who serve identical narrative functions, are keep to a minimum of two.
- The villain is defeated.
- No excessively violent actions occur.

These rules are not entirely inclusive of the rules that would transform the narrative to match the Disney version of Sleeping Beauty – for instance, there is no rule that handles Maleficent transforming into a dragon in Disney version. Additionally, our encoding is on a high enough level that some features aren't considered, such as the individual blessings that the different fairies grant to Sleeping

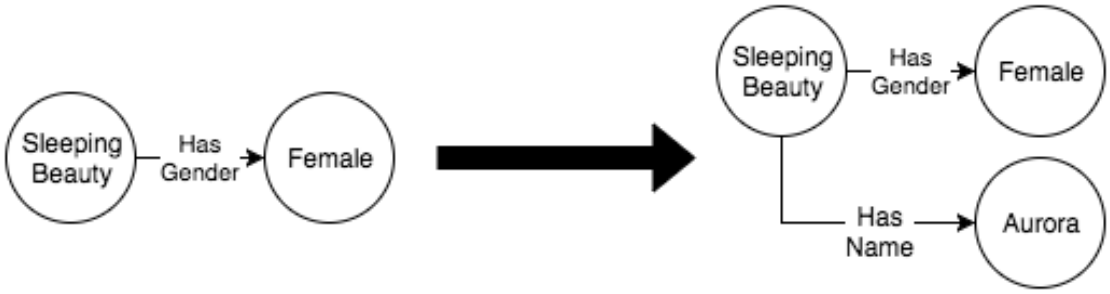

Figure 4.1: A realization of the first rule.

Beauty. However, the point of these rules is not to ensure that the actual Disney script is re-created, but instead are focused on transforming the narrative into one that fits within the style of the Disney.

Figure 4.1 shows an example of the expression of the first rule. In the original narrative, many of the characters are not actually given names: the king and queen, for instance, are only referred to by their titles. To demonstrate how we correct this, we take a character (in this case, Sleeping Beauty) who doesn't have a name. Names are a property of characters, and so to give her a name, we need to create a new fact that says what her name is.

We implement this by having a rule that fires when there is a character who doesn't have a name property. When said rule fires, we pull randomly from a gendered list of names, and create a new fact within the system that says that that character has the chosen name. If the character does not have a gender at this time, we resolve this by randomly assigning a gender and then a coresponding name. In Figure 4.1, we see that the character "Sleeping Beauty" has the name Aurora chosen and assigned.

The animal sidekick rule is similar. If any animal already exists within the narrative, we give that animal the property of sidekick; otherwise we create a new animal with the property in place from the get-go. Afterwards, we then include what being a sidekick means. In this case, the property we wish to include is that

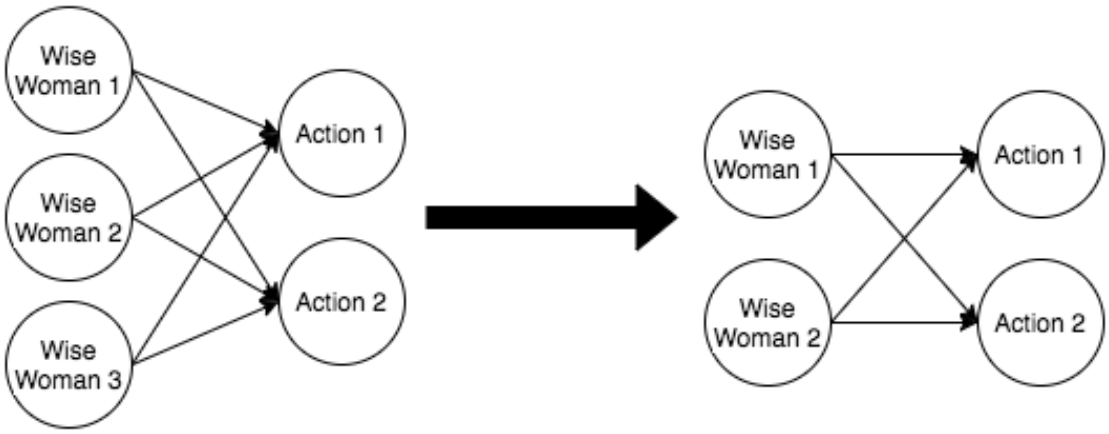

Figure 4.2: Removing an extraneous character.

at each time stamp, the sidekick and protagonist are both in the same location).

Next, we have the repetition reduction. This isn't necessarily an important function – plenty of narratives can feature a large number of characters who all take the same actions. This particular rule was created with the Good Fairies from Disney's Sleeping Beauty in mind. The original Grimm tale instead featured twelve wise woman who filled the same role.

In this, first we need to get a count of the number of characters who all share the same set of actions over the course of the story – specifically, characters who all do the same thing at the same time. When we have three or more characters who fall under that criteria, we select one of them for removal – since all of these characters are considered identically under the narrative, it does not matter which one we select for removal. We then need to remove all of the lingering associations and facts that involve that the removed character. This is shown in Figure 4.2.

For the next rule, we ensure that the villain is defeated. This is requires multiple different checks – if an action in the narrative itself would cause the villain's defeat, then it is a simple matter of appending that particular piece of information. Otherwise, we need to find an action that causes defeat, and have it performed. Most of these actions are performed by a different character (although

there is no reason that the villain couldn't perform the action that causes their defeat), and so we need to ensure that that character is in the correct place in the story to perform that action. Finally, if the action contains any prerequisites, we need to ripple back through the story to ensure that all of them are met before continuing.

In Figure 4.3 we model a transforming part of the story to remove an action that has been labeled within the ontology as too violent. This notion of what qualifies as too violent is subjective, and several Disney movies do feature their villains coming to rather bad ends – in particular, Disney's Sleeping Beauty features Maleficent being stabbed with a magical sword. However, for the purposes of illustration, we have picked that particular scene as too violent, and in need of replacement.

To do this, we first need to find a replacement action that has the same set of consequences – in this case, we're looking for a non-violent action that causes defeat for Maleficent. Our system identifies "Break Weapon" as a replacement for "Stab", and so substitutes that in its place. However, both of the actions have preconditions: while Stab requires a blade, Break Weapon requires that the character has the property Distracting. So, we give the character the Distracting property. If the character has no other actions that require them to have a blade, we can also remove that from the character's list of properties as a tidying measure.

4.6 Discussion and Future Work

While this work has shown reasonable results over the space that its generated in, it is not without limitations. The most obvious is that the Rete rules system used within this system is deterministic; if the same set of conditions and rules are true at any point during execution, the same set of rules with always fire in

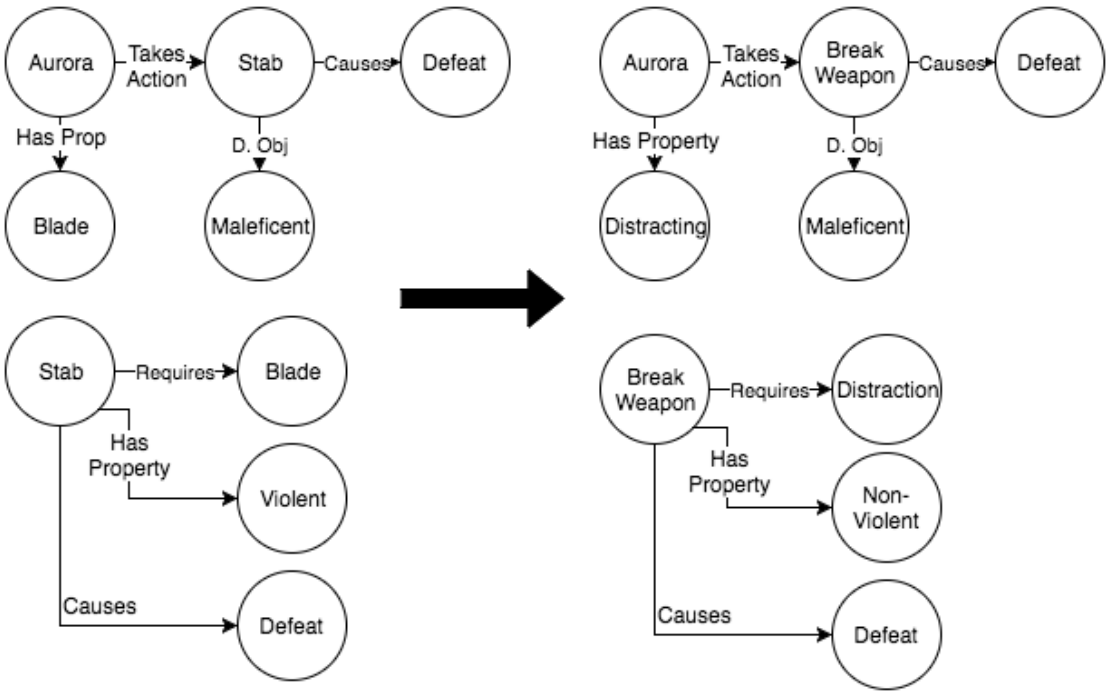

Figure 4.3: Replacing a violent action.

the same order. While this is useful under certain circumstances, for this work it is a limitation. We want to be able to explore the space of possible stories, which we can do by choosing different possible orders for firing rules, or by choosing different facts to match with the preconditions.

Next, the ontology used for generation is hand-authored. While is it certainly possible to have a system learn the ontology of fairy tales (both Disney and Grimm) and use that for generation, we chose for this initial outing to focus on a small, handcrafted ontology.

Finally, the pipeline that this sort of generation would be a part of is not complete. While we are able to manipulate the narrative on the representational level, we do not yet have a mapping from the story representation to natural language.

However, this work does lay the groundwork for the constructing narratives

that fit a particular style, as well as an example of doing so. It additionally provided an example of a formal system for transforming an existing artifact to meet particular stylization goals, which we carried forward into the system presented in the next chapter.

Chapter 5

Solus Forge

5.1 Introduction

The pipeline for creating assets for 3D games is a time consuming and labor intensive process, primarily because most of these assets are handcrafted by experienced 3D modelers. While some of this has been offset by generative tools [40], many hand-designed assets are still required. Because of these, there is a significant benefit in the ability to reuse and repurpose an asset multiple times.

However, this is not always possible; a rock designed for one region of a map may look completely out of place in a different region. There are some simple workarounds; re-skinning and re-texturing the assets allows us to alter the surface-level appearance, and assets can be designed to be stretched and scaled in order to fit various spaces. However, any deeper modifications are often as difficult as simply rebuilding the model from scratch.

To this end, we desire a system capable of generating 3D artifacts from a basic specification that can be procedurally modified to produce stylized variations of that particular artifact. The stylized versions should feature specific, directed changes to the initial artifact such that a minimal set of changes happen to the

initial artifact. Not all styles are going to have a small impact on the structure of an artifact, but by attempting to make minimal changes, we preserve as much of the initial artifact as possible while embodying the new style, meaning that the result is a blend of the two rather than being dominated by the style. However, it is not always trivial to know how a change will affect a given artifact, especially if some of the specification calls for generative components on its own.

In this chapter, we present Solus Forge, a system that allows for the the generation of styled 3D artifacts within a simplified domain, namely the domain of Lego models. Here, we specify our models as hierarchical decompositions with spatial constraints, and then use a two-stage constraint solving approach to generate the output models. In this way, we manage to not only provide generation of models that can be modified by an arbitrary set of rules, but also generate in a way such that we allow for generative components in the specification. Figure 5.1 shows some sample models generated by the system; the two top models are the defaults, and the bottom two are the same models, but incorporating a "Pirate" style. Notice the grey and brown color scheme, as well as the inclusion of the cannon, flag and treasure chest, all strong signifiers of a pirate aesthetic.

These are detailed in the following sections. First we provide our representation framework for the models, then the framework for styles. We then provide a high-level walk through of our design's process, ending with a discussion and a description of future work.

5.2 The Choice of Lego

As covered in Chapter 2, style-centric Procedural Generation is a well-versed field of study. However, most of the visual style systems focus on 2D artifacts, and so we choose to push at this boundary by focusing on 3D artifacts. However,

Figure 5.1: Solus Forge Example Outputs

generating fully-generic 3D meshes is a complex problem, especially when we're trying to utilize the Hierarchical Decomposition model — how do you break a 3D mesh into the different proper parts in a manipulable way, especially when it's unclear where different parts of the model begin or end — in a mesh of a human body, where does the arm end and the shoulder begin?

So, we decided to reduce our domain to one that contains many of the properties we hope to express, while removing the issues involved with general mesh manipulation: Lego models. Legos, in particular, offer a number of useful affordances (as well as some unique challenges) that allow them to provide a test domain for general 3D models. First, they are grid-aligned, which means the rules for how Lego bricks combine are simple as compared to the complexity of possible connections between arbitrary 3D models. Additionally, most bricks can only be attached together by snapping the bottom of a brick to the top of another, further reducing the possible ways they can combine. These discretized and controlled options allow us the ability to utilize technology designed for handling non-continuous problems, as well.

Due to their popularity, and their relative simplicity as compared to arbitrary 3D models, generation of arbitrary Lego structures has been well explored [21, 76, 96]. These systems generate Lego models to fill particular spaces, but, critically, one of the key features they care about is model stability – stability, here, meaning the physical, "will this model actually stay upright if I built it out of actual Lego bricks" sort. From these, and from the sorts of best-practices we see from canonical Lego sets, we derive a short list of useful metrics for creating stable Lego models, such as using the alternating stacks that allow the pieces to hold each other together.

Furthermore, official Lego models are produced in groups of models, each con-

veying a set of @@@FINISH - talk about how Lego models have stylistic languages that run through the different themes and product lines.

Finally, the Lego community is comprised of enthusiasts who make complex and novel builds, and to do so, one of the tools they've developed is the LDraw standard. To quote the main website, LDraw "is an open standard for LEGO CAD programs that allow the user to create virtual LEGO models and scenes" [46]. In particular, LDraw provides an open-source file format, easy to read for both humans and machines, that describes the exact properties of a Lego model in 3D space. We utilize this format as the final output our system.

5.3 Representation of Models

In Solus Forge, Lego models are modeled as a Hierarchical Decomposition, where we build a tree of the model's proper parts[1], until we reach the atomic units of the object. This begs an immediate question: what are these atomic parts for a Lego model?

A naive choice would be to say that each individual Lego piece would be an atomic unit, and certainly, this has merit – each individual piece cannot be subdivided further[2]. While there are a number of cases where we want to consider each individual piece of a model, in cases like the walls of a house, the exact brick layout doesn't matter; rather, we care about the special decorative pieces, while leaving the rest of the walls as generic pieces. Further more, as we transform the model, we may wish to procedurally resize in the model – which, if our model is comprised of a bunch of specific pieces making up the walls, becomes a very difficult problem to handle on the fly. While we could handle that with a large

[1] As explained in Chapter 3

[2] without, at least, the use of force and/or a sharp object

number of non-deterministic nodes, it feels more convenient to model contained groups of generic pieces as a single atomic unit. However, some pieces are more unique and make sense to be handled separately (the pirate cannon, for instance).

To handle these varying cases, Solus Forge has a library of Lego groups, which are simply named collections of pieces. There are two major classes of Lego group: *piece groups* and *volume groups*. Piece groups are the simplier of the two – denoting a pre-specified and fixed set of specific Lego pieces, ranging from a treasure chest to a paired wheel and tire. In constrast, volume groups are made of the generic Lego plates and bricks, and specified as the general volume of space that they are used to fill. This allows us expressive simplisity, since we can refer to (for instance) the walls of a building as being comprised of a particular volume of bricks without worrying about the exact layout of each of those bricks. Secondly, volume groups do not have a predetermined size – they do have a fixed shape that they occupy, such as a rectangular prism, but beyond that, we are given freedom give the size of these groups as a set of ranges, allowing the model to have flexibility in its size and shape. Finally, volume groups are able to intersect with other groups, in a manner evocative of constructive solid geometry – if we have a piece group intersecting with a volume group, we can just remove the parts of the volume group that would intersect with the piece group.

From the library, an author picks out various groups to utilize within their new model. Of course, the Lego groups require a little more than just their identification label – volume groups require size bounds (which is necessary for the generation process, in general), and both kinds of groups can have a color associated with them, which will render all pieces of that group in that color.

Consider the diagram shown in Figure 5.2, which presents the hierarchical decomposition for the car shown in the upper left of Figure 5.1. Each node

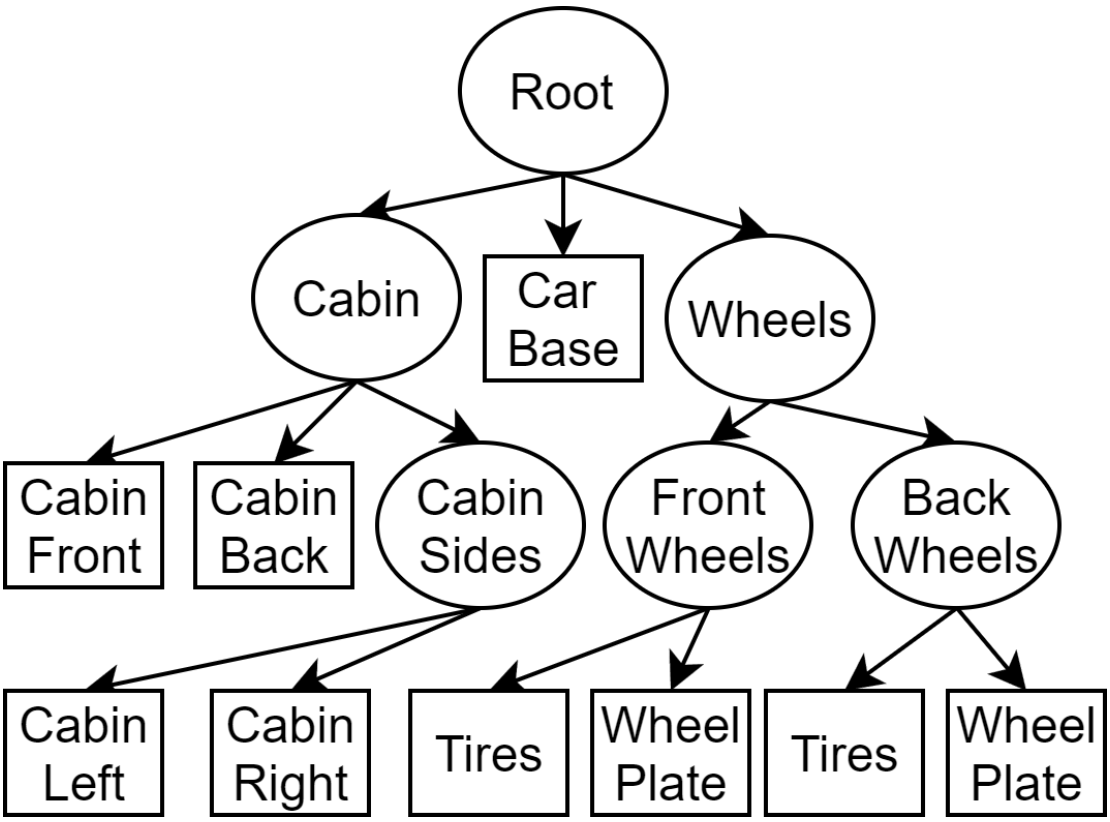

Figure 5.2: Semantic decomposition of the car model

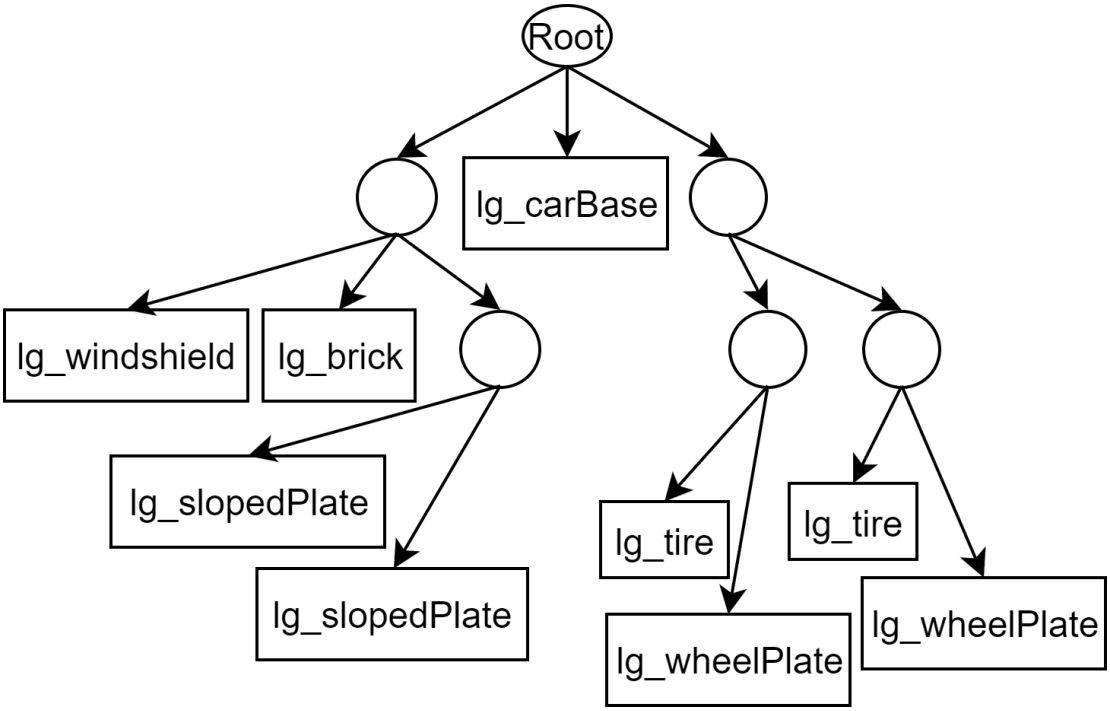

Figure 5.3: Decomposition of the car model with group labels

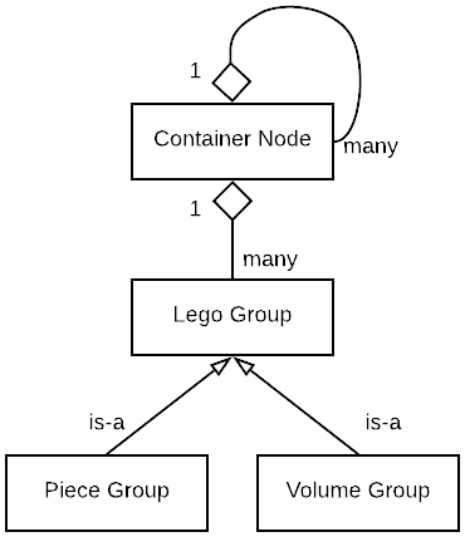

Figure 5.4: A simple UML diagram representing the most of the components of the hierarchical decomposition; only missing the constraint relationships

in the tree is labeled with its semantic meaning (Cabin, Wheel Plate, etc.), to show how this decomposition of the object looks. This structure also includes information about which Lego groups the different nodes of the tree represent, as shown in Figure 5.3. For instance, in this model, we want the front of the cabin ("Cabin Front") to map to a windshield piece and so we specify that that leaf represents "lg_windshield" (the name of the Lego group that represents a windshield piece). The semantic labels have no significance within the system itself other than forming unique identifier for different parts of the object; all of the information needed for generation comes from the Lego groups themselves.

It is worth noting that the labels in Figure 5.2 are not at all considered by the system; these are meant purely as aids for a human designer to structure the model that they're representing to the system. All the system knows is what is contained in Figure 5.3. Think of these labels as variable names in code—the compiler doesn't care if a programmer calls a variable "iteration_counter" or "i", but instead merely operates over the values of those variables.

Additionally, there is an added layer of complexity that we can add to these models: non-deterministic nodes. By allowing a node or a Lego group the ability to be expressed in multiple ways, we create models that are more of a generative space than a singular model, which also supports more complex interactions with styles. Certain choices of the model may be required or prohibited by a style, causing the space of the outputs to be sculpted in interesting ways.

In the models themselves, if we want a node to have a choice embedded into it, then we specify each of its children as being part of those different choices. In this way, we can say that in choice 1, we have two child nodes and in choice 2, we only have one, and so on and so forth—no guarantees of symmetry across the choices are given or necessary. Additionally, we can nest non-deterministic nodes

within other non-deterministic nodes; i.e. picking choice 1 from a node may lead to a new node with a new choice that needs to be made.

Once the primary tree-structure is in place, we now need to add in the positional constraints. These constraints allow us to define where a group is located relative to another. Typically, a group will be located on the surface of another (because of the connections defined by Legos, this is almost always that one group is immediately above or below the other), with constraints on where on that surface the group can be located. A common example of this is placing the wheels of a car not only underneath the main body of the car, but also specifically at the front and back of said car.

Additionally, if two groups are on the same surface, constraints can specify a range for how far apart the groups are allowed to be. Typically this is expressed as ways of keeping distance between two pieces (the couch must be three pegs back from the television), but these constraints also cover containment operations (the couch and television must both be inside the walls) and embedded pieces (a window embedded into a wall).

These positional constraints are not required between every pair of groups, and in some structures, the graph of these relations can be quite sparse. For example, in a given scenario it might be fine if the furniture of a living room is arranged any which way, but only as long as the couch faces the TV. In that case, a single constraint would specify the relative positions of the couch and the TV, with the positions of everything else left unconstrained and free. In some cases, some constraints might be impossible to realize, and in this case, the sketching process will error out.

Within our system, we have the following constraints currently implemented:

- relativePosition – This constraint says that on any given axis (X, Y, or Z),

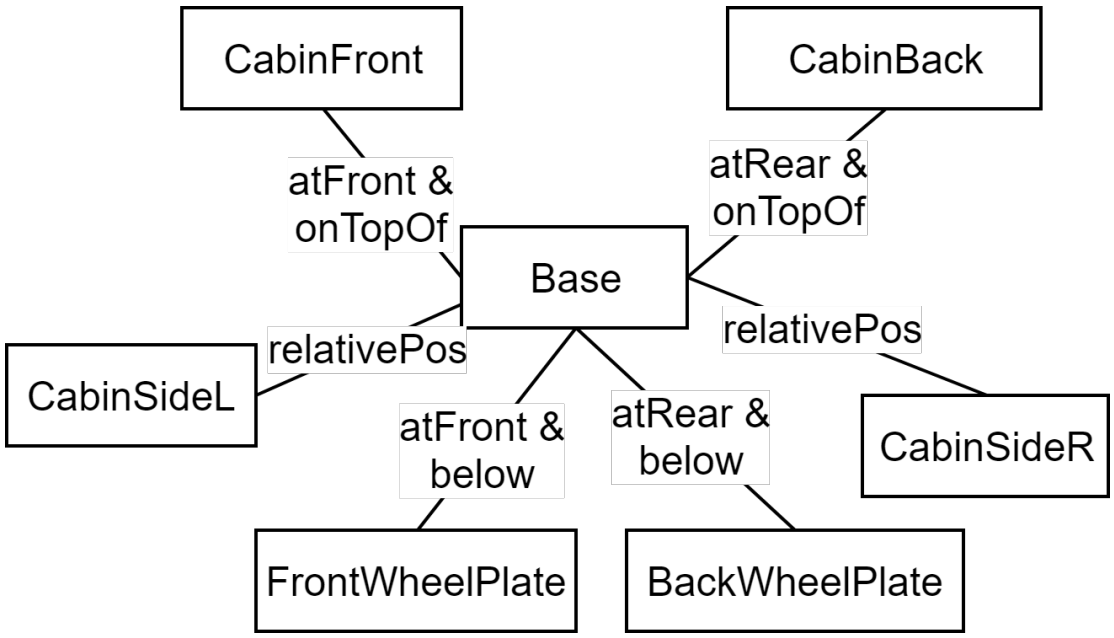

Figure 5.5: Car Part Adjacency Graph

one group's placement is offset from another group's placement by an exact value. This allows us fine-grained control over pairs of groups – such as placing a flag precisely so that its clips are attached to a flagpole, or placing wheels on the pegs of plate that connect them to a car.

- relativeSize – This constraint cares, instead, about the relative sizes of the pair of groups, utilizing the exact same semanics as the above constraint. Here, we get results like "The walls of the house are exactly 6 plates taller than the door to the entrance"

- onTopOf – Ensures that group A is placed directly on the upper surface of group B — in particular, group A's X and Z area intersect with group B's, and that the Y position of group A is exactly on the top of group B's

- below – Conversely, ensures that one group is placed below of another.

- atFront – Given a parameter X, this ensures that group A is within X units

of the front (positive X) edge of group B

- atRear – Conversely, this ensures that group A is within X units of the back edge of group B.

- insetIn – Ensures that group A is inset into the side of group B – giving us the ability to express things like "a doorframe is inset into the walls of a building" This takes two parameters, specifying which side A is inset into.

- distanceApart – Ensures that group A and group B are at least D units apart (using the Taxicab metric)

- inside – Ensures that group A is fully contained inside the boundaries of group B, by constrainting the positions both such that the only valid positions for A are ones make it within B's boundaries on all sided.

So, in our car example, we have the following set of positional constraints, as shown in Figure 5.5:

1. The cabin front and back (represented as simple bricks in the final model) are placed *onTopOf* the car base. They are also *atFront* and *atRear* of the base, respectively

2. The wheels and the plate they connect to are also placed *below* the car base, *atFront* and *atRear* again, and the wheels have a *relativePosition* to the plates.

3. The sides of the car are placed on their respective sides of the model, using *relativePosition* for simplisity.

5.4 Representation of Style

Style is a property of design that is hard to identify or quantify. It can be defined as "a replication of patterning. . . that results from a series of choices made within some set of constraints" [62], which is a useful framework for us to work within. Under this definition, in order to describe a style in an operationalized manner, we first need to describe the constraints that the style places upon the system. With those in place, we claim, the space of choices that the generator can make will necessarily feature the patterning that would make up a style.

While the qualities, choices and insights that lead to the creation of a particular style are an unknown quality, we choose to sidestep that by assuming that the user of Solus Forge has a particular set of constraints that expresses their style already in hand. In order to ensure that any particular style can be represented, we want a representation that errs on the side of being overly general.

To this end, the style is included as a series of model manipulation rules that are resolved during the first phase of generation, ranging from adding on additional parts to replacing entire sections of the model. However, at this point, we run headlong into a problem: In order for our system to make directed style choices to a 3D model, we first need to be able to identify the different parts of that model—after all, the style for the wheels of a car is going to be vastly different that the style for the windows. However, in a generic 3D model, it is not clear where one part of a model ends and the next begins, nor what those parts should be.

To solve this, we leverage that the model is broken up into the well-specified Lego groups mentioned before. Each of these is labeled with a number of properties and the style rules can leverage these built-in labels to indicate whether or not a given rule should apply to a given group.

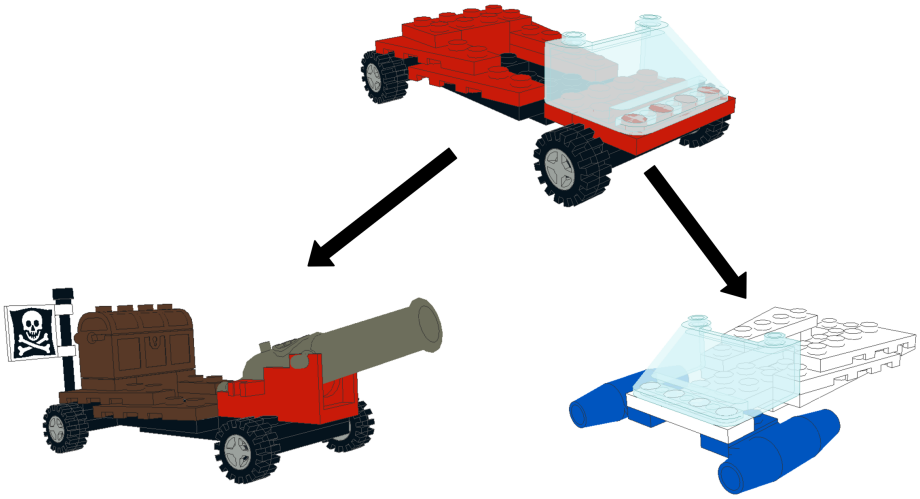

Figure 5.6: Example Style Applications to a Car Model

For the styles, we include the following operations:

- Add in an additional group: The style can add additional Lego groups to the model. In order to do so, it must also include a relational constraint tying the group to another that already exists in the model, some picking a group either arbitrarily or by using the metadata embedded into the groups. A key component here is that we ensure that we only create these new constraints between the groups that are actually present in the final model due to the choices made by the non-deterministic aspects of the model.

- Remove a group: The style can remove Lego groups from the model, identifying the groups to remove either by the name of the group or by a given metadata tag. This necessarily invalidates any constraint that the group is involved with. Removing a group doesn't invalidate a non-deterministic branch of the model that includes said group; the rest of the groups in that choice will be included as normal.

- Replace a group: The style can, instead of removing a group, can replace it with a different group instead, allowing the constraints associated with the old group to carry over to the new group.

- Modify color: We can specify what color of pieces must be used for a given group or type of group; for instance, we can say that a specific piece must be red or that all of the piece groups of a model must be blue.

Consider the pirate style shown in the bottom two models of Figure 5.1. This style is made of the following rules:

- Add in three additional groups: a treasure chest, a cannon, and a flag. In particular, the flag is comprised of three separate pieces (the flag itself, the pole and the connector plate).

- The chest is placed either inside a volume group, or at the back of the model. The flag and the cannon are placed on top of some part of the model.

- Remove: Sloped-piece groups (more as a point of demonstration than any innate pirateness)

- The color scheme is updated to reflect a more wood-and-stone look, focusing on the use of brown and grey.

In contrast, we have a futuristic style (as seen as the second branch in Figure 5.6). This style features the following rules:

- The color of the main body and pieces are set to white, with an exception of thruster pieces, which are turned blue instead.

- We add two thruster pieces to the model, replacing wheel groups, such that one thruster is placed on either side of the Lego model.

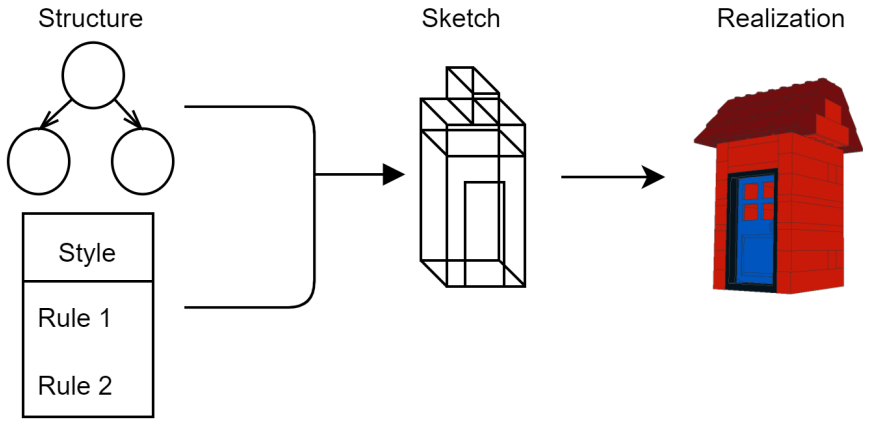

Figure 5.7: Solus Forge System Diagram

As shown in Figure 5.6, these different styles produce variations on the original model that are both immediately recognizable as coming from that model, but also noticeably distinct from each other.

With these styles in place, we are ready to begin the process of generating our models.

5.5 System Overview

Using the 3D modeling approach described above, comprised of a structural decomposition combined with a positional graph, and the style specification, it is now possible to describe the Solus Forge system for applying a style to a 3D model and generating a realized artifact. Figure 5.7 shows the system diagram for this process.

To perform this process, we use a two-step ASP approach by which we first solve for the positions of each of the Lego groups in the model in the Sketching phase, and afterwards solve for the individual Lego pieces in the model in the Realization phase. By separating out the stages like this, we can gain a number of benefits. Firstly, we are able to swap out either part of the system with an

alternate implementation without impacting the other. This allows not only for incremental progress (or bug fixing) on each part independently, but also leaves room to rebuild one of the parts in a completely separate paradigm without impacting the other. Secondly, it gives the affordance of being able to look at and analyze different sketches without realizing them, which allows us to better see how the different parts of the model are interacting with each other.

5.5.1 Sketching

First, we take our model and the style being applied to it, and transform the combination of the two into a sketch of the final output. In this sketch, none of the pieces that actually make up the final model are in place, but the sizes and positions of all of the Lego groups are finalized. Because our constraints can be somewhat loose, there are often a number of different sketches that can be produced from a single model, each with the property that they have a different arrangement of the Lego groups. These differences can be subtle, but they will always be noticeable to an observer.

First, we setup the rules for solving for the new constraint graph, such as the Pirate Car graph seen in Figure 5.8. These rules both cover applying the style as well as handling any non-determinism in the model. In our code base, we call this the Final Graph, comprised of finalChosen Lego groups and finalConstraints that apply between them. We also, additionally, include the rules for determining where all of the chosen Lego groups are in space, such that all of the rules are met and that physical constraints are broken.

While it's useful to frame this as a multi-step process to understand how the different parts work, it's worth noting that all of the sketching process happens conceptually in one step, the solving step. Hence, if the solver looks at a collection

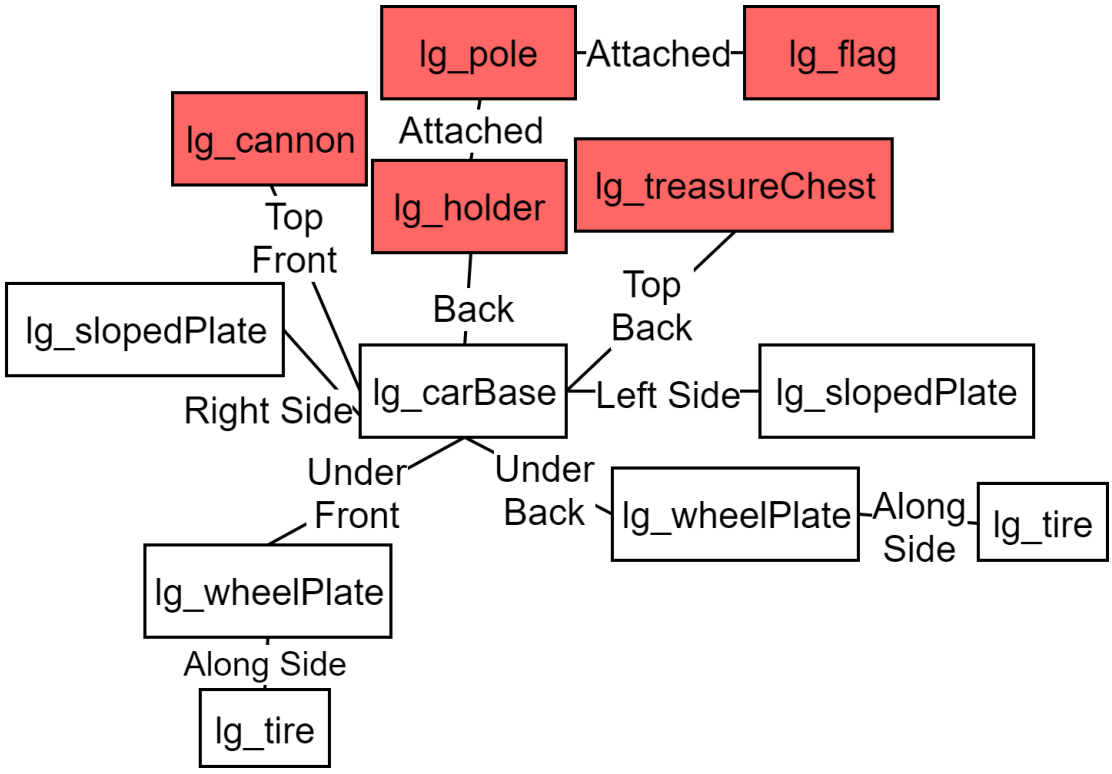

Figure 5.8: Final Car Graph after Pirate Style (Red are newly included groups)

of Lego groups that have no valid physical arrangement, it will mulligan on those and look at other configurations until it either finds one that works or it determines that there is no possible valid solution.

First, let's consider handling the non-deterministic nodes. Since any given node can have one or more choices applied to it, it suffices to simply pick one choice from each node whose parent was chosen. We assume that, by default, the root of the hierarchical decomposition was chosen, and continue down the tree from there.

Next, we look look at the style rules. Color rules are not considered at this stage, and simply pass through the system to be handed off to the next step. Removed groups are flagged as such, and then culled from the final set of chosen nodes (and any constraints involving the removed groups are treated similarly).

Added groups are a bit more complex. Since our model representation requires every group to have at least one constraint connecting it to another group, any added group that our style specifies requires us to also specify a constraint (such that the given group is on one side). In order to add the group in, then, we need to pick a group from the final model to attach the now-added group to.

Finally, replacement rules are a combination of the above. The replaced group or groups are treated as removed, and the replacing groups are treated as added, through with the ease of simply inheriting the constraints from the groups they replace.

With these in place, we then get our final constraint graph; the final representation of all of the groups and constraints required for the final model. At this point, we've reached the representation shown in Figure 5.8. Now, we need to set up our constraint problem to solve for the positions and sizes of each of the groups in space. Our coordinate system is based on the fact that, in general, Legos have

a grid structure – one unit in the X or Z axes is equal to moving one peg, and one unit along the Y axis is equal to the height of a single plate (a brick is exactly three plates tall).

Each of the constraints specified has an underlying logical representation. For instance, a constraint that says that group A is "Under Front" relative to Group B means that group A is under group B (ie, group A's Y-position is such that it ends adjacent to the bottom of B) and that group A is towards the front of B (ie, group A's X position is as far forward as possible, while still being directly underneath B). From these, we able to create a 3D layout (the sketch) of all of the groups.

To that end, our underlying engine for the sketching step takes on a few parts. First, we have some definition references — all of this information is encoded within the system as constants to refer to during generation:

- Space definition - the space in which the pieces can be placed is constructed as a finite range in 3 dimensions
- Piece group definitions - Each piece group has a fair bit of associated data - each piece group needs to have its size specified, and all have metadata that is leveraged by the styles.
- Volume group functions - Since volume groups can be of any size, we include rules for choosing their size here. Some of these choices are straightforward (a rectangular wall can simply pick a size in each dimension), but others can be more complex, such as the sloped roof we see in the building model. For that, we specify the position, size and orientation of each of the rows of bricks based on the size and location of the footprint of the group overall. This turns one simple group into a large number of individual groups for the purposes of further generation.

We then have a set of rules for handling the constraints. Since constraints can come in a number of different varieties, we have a number of different rules for handling the various kinds of constraints. For instance, all positional constraints follow a similar pattern, since they all constrain the place of one group relative to the position of another. For example, assume that we have group A on top of group B. In this case, we fix group A's vertical position to be exactly adjacent to the top of group B (requiring group B's vertical position and height to be known, of course), and restrict group A's position on the other two axes to intersect with group B's.

Finally, for each group, we pick its location in space. Since each group is related to others by constraints, we have a number of restrictions on the placement of any given group. However, often times, there are axes along which we have free choice within a range — for instance, we don't care where the cannon goes, as long as it's on top of the other group. Because of this, there is room for variation in the different sketches that are produced. Many of these sketches will look similar to each other, having a group's placement differ by a single peg, but they will all be unique and differ from each other in this regard.

These rules[3], when combined with a model and style specification, produce a sketch, which is captured both in a human readable JSON file, as well as a long list of facts that is handed off to the the Realization step.

5.5.2 Realization

Once all of the positional information is in place, the system then needs to realize the sketch. For our Pirate car, this step is trivial, since all of the Lego groups included are piece groups. As such, the only step that this step takes is

[3]For a full description of the rules, please view the source at `https://github.com/jomazeika/SolusForge`

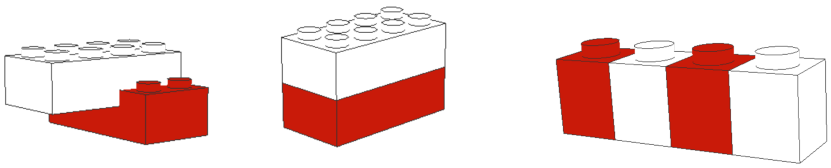

Figure 5.9: Three examples of prohibited brick placements in the realizer. From left to right, we have two pieces intersecting, two identical pieces stacked on top of each other, and four pieces placed in a way that could be converted into a single piece.

to specify the particular pieces that make up the different groups in the locations determined by the previous step. However, for the house model, we need to a fair bit of solving to construct the walls of the model, and the roof of the base model.

For this, we specify a list of possible pieces—for the walls, this is a set of basic Lego bricks and plates, and we also include a set of sloped pieces for the roof—as a list of tuples, denoting size of each piece. We also include our information from the sketch, transforming into a set of voxel coordinates that need to be filled as a list of tuples—including the space of each volume group, then subtracting out the space encapsulated by the piece groups.

We then solve for a collection of pieces (and their locations) such that the following criteria are met:

- The target volume, and only the target volume, is covered with pieces.
- No two pieces used intersect with each other
- No two identical pieces are directly on top of each other (decreasing the potential for free-standing sections of the model)

- The minimal number of pieces are utilized in creating the model, again increasing stability.

Visual examples of these three constraints can be found in Figure 5.9.

While a large number of solutions exist that meet the above criteria, we (from previous Lego literature) know that using the fewest number of Lego pieces in our final model will help increase the overall stability. So, we include a statement to optimize our model to use the fewest pieces possible. Our final model is pictured in Figure 5.1.

However, this particular framework is not without its limitations. As the number of pieces required to fill in the space increases, the time that the problem takes to verify the previous steps increase exponentially since each piece's position is considered independent of all of the other pieces'. To compensate for this, we again split our solving process in half, this time introducing some assumptions that, while producing interesting and useful results, introduce a level of generative weirdness to the outputs.

To this end, the process is split into halves; in the first, we solve for the layers of the final output, and in the second, the locations of the pieces within those layers. In this way, we turn a three dimensional layout problem into a two dimensional problem where solutions can be reused. Here, we say that layers are *identically shaped horizontal cross-sections of the area that will be filled in with pieces.*

As with the sketching phase, all of the solving steps happen in an order specified by the constraint solver, rather than a simple pipeline. However, when organizing the system, we specified a few clear

- We identify the cross-sections of the space that are identical to each other, and partition those into layers of either height 1 (the in-model height of a Lego plate) or 3 (the height of a Lego brick)

- In order to refer back to a given layer, and to get a count of the number of unique layers, we pick a set of *exemplars* for each layer.
- We also find an *Exemplar map* that specifies which layers correspond with which exemplar.

Once we have the layers, the process of laying out bricks is similar to before, with the exception that each layer is considered separately, rather than the space being seen as one whole problem. We solve for two different layouts for each layer. By solving for the two different incarnations, we can alternate between them in the final model, and thereby ensure that the model doesn't feature identically sized pieces on top of each other.

5.6 Conclusions

In this chapter, we have presented three major contributions. First is a framework for structuring 3D Lego models as a hierarchical decomposition of their component parts, including generative components, as well as the positional relationships between these different components. Second, a declarative style specification that allows us to describe the particular transformations required to create a new model in the specified style. Finally, we showed a two-phase constraint logic programming pipeline and realization of this system that allowed us to produce transformed Lego models utilizing the previously mentioned frameworks.

Chapter 6

Answer Set Programming

6.1 Introduction

In our previous chapter, we described Solus Forge, a system for generated stylized Lego Models from a Hierarchical Decomposition (defined in Chapter 3) and a style specification. In this chapter, we go into the details of the implementation of Solus Forge, which is built using a declarative form of programming called Answer Set Programming (ASP).

In this chapter, we will talk about ASP, giving a quick overview of the concepts and constructs involved, as well as its uses in other procedural generation systems, Following that, we will dive into specific implementation details for Solus Forge, first covering the Sketcher before talking about the iterations that the Realizer passed through to reach its current form.

6.2 Introduction to ASP

Answer Set Programming (or ASP) "is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems" [57]. Given a

problem, described as a space of values and a set of constraints on those values, an ASP system will solve for one or more set of those values that satisfy all of the different constraints. For Solus Forge, we utilize clingo, a combination of gringo (a grounder) and clasp (a solver) produced by the Potassco group [33], which takes in input using the language AnsProlog [16], a Prolog-like language for describing ASP.

Most modern ASP systems solve the problem in two phases, grounding and solving. Here, the solver is a modified version of a SAT-solver, which looks for particular combinations of variables that satisfy some series of boolean constraints. To do this, however, the solver requires that the input to the system be in a very specific format – namely, that all of the different values that could possibly exist within the final solution be spelled out, so that the solver can experiment with various truth values for these variables. The grounder's job is to *ground* the AnsProlog file (which, as we'll see in the next section, contains a lot of variables and syntactic sugar) into a form that the solver can operate over. The solver then takes the grounded version of the program and finds one or more sets of literals that satisfy all of the constraints of specified within the program.

While there are many other off the shelf constraint solving systems that we could have alternatively chosen, clingo provided a number of useful advantanges. First, and most critically, clingo has an academic history of use for procedural generation (see Section 6.3 for the literature survey), which gave us an existing community to validate within. Secondly, clingo (and ASP broadly) differ from other constraint solving systems in that, by default, they offer multiple solutions to the constraint problems, rather than only providing the first one the system finds. This allows us to rapidly explore a space of outputs, and produce multiple answers without hacky solutions to force the solver to explore alterate spaces and

multiple solving steps.

6.2.1 ASP Syntax

ASP programs are comprised from three major components: Facts, Rules and Constraints. Facts (also called literals) are declarative statements that instantiate some fact to exist within the solution (also called the universe). These can have nested parameterization, contained in parens, and ranges. All of these are given to exist in the world, and give us the start point and universal truths for the solver to operate over. Below, we have some examples of facts:

```
safeMode.
person(stella).
person(noemie).
triangle((2,2),(0,0),(1,2)).
startingLocations(0..15).
```

Rules are statements that derive new facts from existing ones. In particular, rules consist of a left-hand side, the head, and a right-hand side, the body: If there are sets of literals that match the body of a rule, the facts described in the head fire. For instance:

```
isMorning :- hour(9).
```

This rule says that if the literal "hour(9)" exists, we derive the fact "isMorning"[1]. These rules can also feature variable declarations, which match any literals that exist in their slot. For instance:

[1]The interpretation, here, of course, being that, if it's 9 AM, it's morning

```
accessible(X) :- chosenStart(X).


chosenStart(1).
chosenStart(2).
chosenStart((3,5)).
chosenStart(seaShrine).
```

The rule at the top of the block will match with all four of the "chosenStart" literals, producing the following facts in addition to the four that already exist:

```
accessible(1).
accessible(2).
accessible((3,5)).
accessible(seaShrine).
```

Finally, we have constraints. Constraints are written as rules with no heads, and will invalidate any answer set that contains their body[2]. Examples:

```
:- unhappy(jim).
:- tile(T), inaccessible(T).
```

The first constraint is obvious. If the fact "unhappy(jim)" exists in a particular answer set, we prohibit that answer set from being The second one says that if there's any pair of "tile(T)" and "inaccessible(T)" facts, specifically where the T's match, destroy the universe.

However, these rules are insufficient to give us the full range of expression we need. Since all rules fire on all facts that they match, there's no room for multiple, different answers under the formulation thus far – the output solution

[2]or, as I like to say, "will destroy the universe"

is completely deterministic based on the input. This is where *Choice rules* come into play. Choice rules feature the following syntax:

```
{meetingDay(D) : openDays(D)} = 1
1 <= {openDays(D) : workDays(D)} <= 5
```

The first rule here says "given all of the openDays facts that exist in the universe, pick exactly one and create a corresponding meetingDay fact." In short, we're making a choice of all of the openDays facts to pick a meetingDay. The second rule is very similar, except it picks between 1 and 5 (inclusive) workDays facts to turn into openDays instead.

```
{pair(I,L) : location(L)} = 1 :- item(I).
{pair(I,L) : item(I)} = 1 :- location(L).
```

Here, we have a construction that's used to great effect in Solus Forge. The first translates to "for each item(I), pick exactly one location(L) and create a pair(I,L) fact" with the mirrored version existing in the second rule. In this way, we create a set of pair facts that are a bijection between locations and items, where each location and item appear in exactly one pair.

With these basic constructs, we are able to setup a generative space and sample within it. Specifically, the choice rules allow us to create an entire possibility space (all choices are initially valid) and we can pick as we will between all of the different options. Of course, some of these choices violate one constraint or another and we need to search through the space of all these solutions to find the ones which satisfy all of the different constraints. In this way, constraint solving (and ASP in particular) form a subtractive generative process, one where the initial domain of possiblilities is much larger than the set of valid possibilities. This is in constrast

to techniques like grammars, where the only valid combinations of choices are makeable.

Since clingo accepts multiple input files in a call (simply concatenating them) and because ASP programs are completely line-order independent, a common practice is place all of the rules and constraints together, in a single file, while different files containing differing sets of facts can be swapped in or out to form the basis for different problems. Addition introduction to this paradigm and its concepts can be found here [67].

6.3 ASP for PCG

Smith and Mateas wrote the first summary of utilizing ASP for Procedural Generation [86] — in this article, they cover a similar summary of ASP as a whole, an example of utilizing ASP to generate a chromatic maze, and a few oother case studies from other sources of utilizing ASP for generative methods. One example, DIORAMA, is a map generator for a real-time strategy game, another is Variations Forever, an educational puzzle game for children [85]. However, since then, a number of other examples have emerged, from map generator for Rougelike games [87] and action-adventure games [89], to system that created poems using a formalized structure [98].

With this space of systems, it's clear that we have both an eye towards reasoning over spatial layouts — a system for operating over 3D spaces to produce game levels also exists in the literature [5]. ASP, of course, is by no means the only constraint solving paradigm used in generation, as evidenced by the vast body of literature focused on using constraint solving for space layout problems [60, 78].

6.4 Solus Forge Implementation

As discussed in the previous chapter, Solus Forge is a system for generating stylized 3D Lego models in a two-stage process: a *sketching* phase and a *realization* phase. In the sketching phase, we take in the hierarchical decomposition of the model and solve both the stylization and placement of the Lego Groups – the high-level collections of Lego pieces that Solus Forge operates over. In the realization phase, we take a sketch and then solve for the positions of all the individual bricks that make up that sketch.

This split is important both on a conceptual and technical level. On the technical side, breaking up the solving into these phases allows the system to be more performant, since it will be solving for fewer overall positions than a combined version would require (and, as we'll see later, a number of additional features had to be included in the realizer to make it performant). On the conceptual side, since the realizer solves for different piece layouts for Lego Groups that are already in particular places, there's no real difference between two different solutions. The meaningful differences in model are going to be at the sketcher level, where we see different Lego groups being placed in different locations on the final model. While some of these can be trivial, they are at least easily perceptible to the human eye. So, when we're looking at different solutions, we are in particular, looking at different sketches of the model.

In addition to the ASP core, Solus Forge also features some Python scripts that handle file IO, convert the final realized model into the LDraw Standard[3], and open source format for specifying Lego bricks and models.

However, before we dive into the details of the different phases of the solver, we first should discuss our data structures for the inputs.

[3]As specified here: https://www.ldraw.org/article/218.html

6.4.1 Model Specification

As mentioned in the previous chapter, the models in Solus Forge are represented in the Hierarchical Decomposition format, and so we need to encode structures of this format into ASP.

To do so, all models begin with an implicit node called Root that forms the basis of the model. This allows us to ensure a consistent starting point for the model, allowing us to always start generation from the same point. On the other hand, it does mean that "Root" is pre-used as a node label.

Internal nodes utilize have the format:

```
node(name(N), parent(P,ND)).
```

where N is a label/name for the node (all of which need to be unique), P is the name of the node's parent, and ND is a non-deterministic branch label. In particular, if a parent has children who have different non-deterministic branch labels, only one of the branches will be expressed in any given solution. These nodes do not need to carry any further information, as they are merely the trellis for the Lego groups to hang on.

Speaking of Lego groups, we encode those in the tree as:

```
legoGroup(name(N), piece(L), parent(P,ND)).
legoGroup(name(N), volume(L, params(F)), parent(P,ND)).
```

Here, name(N) and parent(P,ND) are identical to how they're utilized in the internal nodes. The big difference is the middle parameter, which (simply put) specifies what Lego group is being represented. Volume groups, in particular, require extra parameters to be specified — usually, parameters around the size of the object.

6.4.2 Style Specification

As mentioned in the previous chapter, there are four different classes of style rule that we have implemented within Solus Forge: add a new subtree, remove a group or groups, replace a group or groups with another subtree and apply a color to a group or groups.

These are specified within the system as follows:

- Add group: We add in groups using styleAdd(Node, Relation, Params, Fixed) rules, where Node is a specified name of a sub-tree, Relation is the new constraint we attach the focusGroup (an additional fact we need to specify, labeling one of the groups in the subtree Node as the "main" group of the model) to the main graph with. The Params parameter is the meta-data for the constraint specified, and Fixed is either *fixed* or *opt* — where fixed means that the constraint has to apply, and opt means that it is optional. We can have multiples of these applying to the same subtree, but with different constraints, some of which are required and some of are optional.

- Remove group: These are specified as remove(Class, Handle) facts, where Handle is *remove*[4] and Class is a met-data label that we use to match. Specifically, this says that any group that matches the specific label will be removed from the model.

- Replace group: Here, we specify that we want to replace a particular set of groups with a specified sub-tree. These facts are described as styleReplace(Target, New), where Target is similar to the the Class parameter in remove(), and New is a sub-tree as in styleAdd().

[4]This is left as a parameter for future work for other ways of handling orphaned groups

- Color Specification: The simplest of these facts – color is specified purely as colorRef(Type,Color) facts, where Type is a group meta-data label, and Color is a lego piece color — the system will make sure that that group features the given color in the final output.

6.5 Sketcher Implementation

For the Sketching phase of Solus Forge, our intend is to take in the model description and the style description, and utilize them to produce what we are calling a *sketch*. Specifically, this sketch is a list of groups in the final output of the model, with their finalized positions, sizes and colors.

To do these style transformations, there are a few moving parts the overall sketching system needs to address – first, we need to apply the style rules to the model. Then, we need to handle our non-deterministic choices. Finally, we need to place and size all of the groups such that none of the constraints in the model are violated. Of course, since the ASP solving process happens in effectively a black box, we can't specifically state that these happen in any given order. However, it is useful to group the logical flow of the system into these phases for the sake of the humans involved with the system.

As another helper for the humans, we also break up this system into a number of different files, each corresponding to a different core chunk of logic. As we explain the details of the Sketcher, we will be walking through file-by-file[5].

6.5.1 applyStyle.lp

Here, we have the rules for transforming the model based on the style rules provided. These are broken up into four distinct sections, one for each of the

[5]All of the raw ASP files utilize the .lp file extension

style transformation rules mentioned in the previous chapter. As stated above, these start as styleAdd(Node, Relation, Params, Fixed) facts included in the style specification. From there, it adds all of the subtree to the model, as well as including the newly specified required constraints, and choosing some subset of the optional constraints to be included as well.

```
%Create the new pieces of the model
{node(name(Node), parent(P,1)) : nodeTarget(name(P))} = 1
   :- styleAdd(Node,Relation,Params,Fixed).
legoGroup(N, piece(T), parent(Node,1))
   :- groupRef(N, type(T), parent(Node)), node(name(Node), P),
   styleAdd(Node,Relation,Params,Fixed).

%choose the optional constraints to include and the required ones
includeCons(Node,Relation,Params)
   :- styleAdd(Node,Relation,Params,fixed).
{ includeCons(Node,Relation,Params) }
   :- styleAdd(Node,Relation,Params,opt).

%Pick a second group to attach to the constraint.
{chosenRelation(Relation, Target, name(Group), Params) :
   groupTarget(Target), Target != Group} = 1
   :- includeCons(Node,Relation,Params), focusGroup(Node, Group).

constraint(Relation, Group, Target, Params)
   :- chosenRelation(Relation, Target, Group, Params).
```

The remove rule functionality is a little bit simpler – here, we simply mark

the groups and constraints for later removal. Additionally, here we include the fact *remove(“”,“”)* – this is an artifact of how ASP handles missing rules; it's merely present to prevent the solver from returning unsat when no remove facts are included.

```
remove("","").
removedGroup(Name) :- remove(T,Flag), isA(Group,T),
   groupType(Name,Group).
removedCons(T,N1,N2) :- constraint(T,N1,N2,P),
   remove(Type, remove), groupType(N1,Group), isA(Group,Type).
removedCons(T,N1,N2) :- constraint(T,N1,N2,P),
   remove(Type, remove), groupType(N2,Group), isA(Group,Type).
removedGroup(N1) :- constraint(T,N1,N2,P), removedGroup(N2).
```

For replacement, we start by re-using the remove() logic we established above on the groups that we wish to replace with other ones. We then also keep track of the constraints associated with the replaced groups.

```
remove(Target,remove) :- styleReplace(Target,New).
replacedGroup(Name) :- styleReplace(T,New), isA(Group,T),
   groupType(Name,Group).
replacedCons(T,N1,N2,P) :- constraint(T,N1,N2,P),
   styleReplace(Type,New), groupType(N1,Group), isA(Group,Type).
replacedCons(T,N1,N2,P) :- constraint(T,N1,N2,P),
   styleReplace(Type,New), groupType(N2,Group), isA(Group,Type).
```

We can't assume that we will always be able to replace every group for every rule – for instance, we might have replacement rules that would apply to the same node twice. To allow for this, we focus on ensuring that as many targets are

replaced as possible. This is left as a soft constraint as well, because in some cases, we might not be able to replace all of the groups that a constraint applies to.

```
replaceNode("","").
{replacementNode(New,GroupName) : replacedGroup(GroupName)} <= 1
   :- styleReplace(Target,New).
:- replacementNode(N1,G), replacementNode(N2,G), N1 != N2.
chosenReplaced(GroupName) :- replacementNode(New,GroupName).
missedReplace(GroupName) :- replacedGroup(GroupName),
   not chosenReplaced(GroupName).
:~ missedReplace(G). [1@3,G]
```

Once we've decided on the set of new nodes to add in place of the replaced ones, we need to add in the new sub-tree to the model. Here, we take the build out the internal nodes.

```
newReplacementNode(name(New),parent("Root",1)) :-
   replacedGroup(GroupName), legoGroup(GroupName,T,P),
   replacementNode(New,GroupName).
newReplacementNode(name(Node), parent(P,1)) :-
   replaceNode(Node,P), newReplacementNode(name(P),P2).
node(name(Node), parent(P,1)) :- replaceNode(Node,P),
   newReplacementNode(name(P),P2).
node(name(Node), parent(P,1)) :- replaceNode(Node,P),
   node(name(P),P2).
node(N,P) :- newReplacementNode(N,P).
```

Then, we construct the new Lego groups.

```
newLegoGroup(N, T, parent(Node,1)) :-
   groupRef(N, T, parent(Node)),
   newReplacementNode(name(Node), P).
legoGroup(N,T,P) :- newLegoGroup(N,T,P).
```

Finally, we add in the constraints from the original (now removed) group. These are applied to the focus group (borrowing the concept from the add logic), and we make sure that each of the new groups we incorporate includes at least one constraint connecting it to the original model as a sanity check.

```
possibleConstraint(T,name(Focus),N2,P) :- replacedCons(T,N1,N2,P),
   replacementNode(New,N1), focusGroup(Focus, New),
   not removedGroup(N2).
possibleConstraint(T,N1,name(Focus),P) :- replacedCons(T,N1,N2,P),
   replacementNode(New,N2), focusGroup(Focus, New),
   not removedGroup(N1).

replaceGroupCheck(name(N)) :- replacementNode(New,N2),
   focusGroup(N, New).
chosenReplaceConstraint(T,N1,N2,P) :- possibleConstraint(T,N1,N2,P).

replaceConst(N) :- chosenReplaceConstraint(T,N,N2,P),
   replaceGroupCheck(N).
replaceConst(N) :- chosenReplaceConstraint(T,N1,N,P),
   replaceGroupCheck(N).
:- replaceGroupCheck(N), not replaceConst(N).
constraint(T,N1,N2,P) :- chosenReplaceConstraint(T,N1,N2,P).
```

Finally, we have our simplest style rules – the color rules. These simply keep track of which groups are to be rendered in which colors, feeding this information forward until we get to the final rendering step.

```
colorRef(N1,C) :- node(name(N1),parent(N2,_)), colorRef(N2,C).
colorRef(N,C) :- legoGroup(name(N),_,parent(P,_)), colorRef(P,C).
```

6.5.2 resolveND.lp

Then, we have resolveND.lp, the file with all of the logic for handling non-deterministic nodes. As described in the previous chapter, when we have two or more subtrees to choose between in a model, we can simply pick one or the other, so long as the other constraints in play are not violated.

```
nodeChoice(Name,Id) :- node(Node, parent(Name,Id)).
nodeChoice(Name,Id) :- legoGroup(N, T, parent(Name,Id)).
{ndChosen("Root",Id) : nodeChoice("Root",Id)} = 1.
{ndChosen(Name,ID) : nodeChoice(Name, ID)} = 1
   :- node(name(Name), parent(PName,PID)), ndChosen(PName,PID).

%Force style pieces to appear
:- nodeRef(name(N)), not ndChosen(N,1).

{chosenPiece(Name,piece(Group), parent(PName,PID)) :
   isA(Group,ClassName) } = 1
   :- ndChosen(PName,PID), legoGroup(Name, piece(ClassName),
   parent(PName,PID)), not removedGroup(Name).
chosenVolume(Name,volume(Type,Params),parent(PName,PID)) :-
```

```
    legoGroup(Name,volume(Type,Params), parent(PName,PID)),
    ndChosen(PName,PID), not removedGroup(Name).

:- removedGroup(N), chosenPiece(N,V,P).
:- removedGroup(N), chosenVolume(N,V,P).
```

6.5.3 sketcher.lp

Afterwards, we have the confusingly named sketcher.lp, which handles a bit of tidying up from the previous pieces of logic – bubbling down constraints to the base on the nodes, and creating a final set of groups to reason over the positions for. The bubbling-down effect (applying constraints to the child groups of internal node) is especially important for the nondeterministic nodes, since we need to make sure that the constraints apply to the various choices.

```
finalChosen(N,volume(T,Params))
   :- chosenVolume(N,volume(T,Params),parent(P,V)),
   ndChosen(P,V), not removedGroup(N).
finalChosen(N,T) :- chosenPiece(N,T, P), not removedGroup(N).

constraint(C,New,N2,P) :- constraint(C,name(N1),N2,P),
   node(name(N1),Parent), node(New,parent(N1,PID)),
   ndChosen(N1,PID).
constraint(C,N1,New,P) :- constraint(C,N1,name(N2),P),
   node(name(N2),Parent), node(New,parent(N2,PID)),
   ndChosen(N2,PID).

constraint(C,N1,Group,P) :- constraint(C,N1,name(N2),P),
```

```
   legoGroup(Group,Type,parent(N2,V)),
   ndChosen(N2,V).
constraint(C,Group,N2,P) :- constraint(C,name(N1),N2,P),
   legoGroup(Group,Type,parent(N1,V)), ndChosen(N1,V).
```

The other thing we have here is the check that, after all of our transformations to the core model, the constraint graph is still fully connected.

```
finalConstraint(C,N1,N2,P) :- finalChosen(N1,T1),
   finalChosen(N2,T2), constraint(C,N1,N2,P).


cGraph(N) :- anchor(N).
cGraph(N1) :- finalConstraint(C,N1,N2,P), cGraph(N2).
:- finalChosen(N,T), not cGraph(N).


:- constraint(C,N,_,_), legoGroup(N,piece(Type),Parent),
   badConstraint(C,Type,1).
:- constraint(C,_,N,_), legoGroup(N,piece(Type),Parent),
   badConstraint(C,Type,2).
```

6.5.4 constraint.lp

Next, we have constraint.lp, which has all of the logic for all of the implemented constraints within the system; namely, the ones mentioned in Chapter 5. Here, we skip over some of the implementation where things are identical, up to a direction (for instance, onTopOf and below are effectively the same constraint, but applying to a different side of the same model). So, let's break these down, starting with the onTopOf constraint we mentioned.

This constraint positions one group to be physically on top of another – which makes setting the y-position of the group easy. The x and z positions of the group are a bit trickier, as we need to ensure that the two groups exist within each others boundaries along those axes in particular.

```
groupPos(N1,y,P2+S2) :- finalConstraint(onTopOf,N1,N2,P),
   groupPos(N2,y,P2), groupSize(N2,y,S2), P2 <= maxY,
   P2 > 0.
topInter(N1,N2,Axis) :- finalConstraint(onTopOf,N1,N2,P),
   notUp(Axis), groupSize(N1,Axis,S1), groupSize(N2,Axis,S2),
S1 -1 >= #sum{P2 : groupPos(N2,Axis,P2); -1*P1
   : groupPos(N1,Axis,P1)},
1 - S2 <= #sum{P2 : groupPos(N2,Axis,P2); -1*P1
   : groupPos(N1,Axis,P1)}.
topInter(N2,N1,Axis) :- finalConstraint(onTopOf,N2,N1,P),
   notUp(Axis), groupSize(N1,Axis,S1), groupSize(N2,Axis,S2),
S1 -1 >= #sum{P2 : groupPos(N2,Axis,P2); -1*P1 :
   groupPos(N1,Axis,P1)},
1 - S2 <= #sum{P2 : groupPos(N2,Axis,P2); -1*P1 :
   groupPos(N1,Axis,P1)}.

:- finalConstraint(onTopOf,N1,N2,P), not topInter(N1,N2,x).
:- finalConstraint(onTopOf,N1,N2,P), not topInter(N1,N2,z).
```

Next, we have atFront and atRear, which force one group to be within a specified number distance from the front or rear of the other.

```
{groupPos(N1,x,P2..P2+X-1) } = 1 :-
```

```
   constraint(atFront, N1, N2, params(X)), groupPos(N2,x,P2),
   P2 <= maxX, P2 > 0.


{groupPos(N1,x,P2+S2-X+1..P2+S2) } = 1 :-
   constraint(atRear, N1, N2, params(X)), groupPos(N2,x,P2),
   groupSize(N2,x,S2), P2 <= maxX, P2 > 0.
```

insetIn, as described, sets a piece group into the wall of a volume group – for things like doors, windows and other assorted decorations.

```
groupOrientation(N1,Axis) :-
   finalConstraint(insetIn,N1,N2,params(Axis,Side)).
{ groupPos(N1,Axis,P2..P2+S2-S1) } = 1 :-
   finalConstraint(insetIn,N1,N2,params(Axis,Side)),
   groupPos(N2,Axis,P2), groupSize(N2,Axis,S2),
   groupSize(N1,Axis,S1), P2 > 0, P2 <= M, max(Axis,M).
groupPos(N1,A2,P2):-
   finalConstraint(insetIn,N1,N2,params(Axis,"-")),
   axisPair(Axis,A2), groupPos(N2,A2,P2),
   P2 > 0, P2 <= M, max(A2,M).
groupPos(N1,A2,P2+S2-S1):-
   finalConstraint(insetIn,N1,N2,params(Axis,"+")),
   axisPair(Axis,A2), groupPos(N2,A2,P2), groupSize(N2,A2,S2),
   groupSize(N1,A2,S1), P2 > 0, P2 <= M, max(A2,M).
{ groupPos(N1,y,P2..P2) } = 1 :- finalConstraint(insetIn,N1,N2,P),
   groupPos(N2,y,P2), groupSize(N2,y,S2), groupSize(N1,y,S1),
   P2 > 0, P2 <= M, max(A2,M).
```

Next, we have relativePos and relativeSize, which set one group's position or size to a value relative to the other group's.

```
groupPos(N1,x,P2+X) :- groupPos(N2,x,P2),
   finalConstraint(relativePos,N1,N2,params(X,Y,Z)),
   X != null, P2 <= maxX, P2 > 0.
groupPos(N1,y,P2+Y) :- groupPos(N2,y,P2),
   finalConstraint(relativePos,N1,N2,params(X,Y,Z)),
   Y != null, P2 <= maxY, P2 > 0.
groupPos(N1,z,P2+Z) :- groupPos(N2,z,P2),
   finalConstraint(relativePos,N1,N2,params(X,Y,Z)),
   Z != null, P2 <= maxZ, P2 > 0.

groupSize(N1,x,P2+X) :- groupSize(N2,x,P2),
   finalConstraint(relativeSize,N1,N2,params(X,Y,Z)), X != null.
groupSize(N1,y,P2+Y) :- groupSize(N2,y,P2),
   finalConstraint(relativeSize,N1,N2,params(X,Y,Z)), Y != null.
groupSize(N1,z,P2+Z) :- groupSize(N2,z,P2),
   finalConstraint(relativeSize,N1,N2,params(X,Y,Z)), Z != null.
```

Next, we have the distanceApart constraint - which ensure that the two groups in question are at least C units of distance apart.

```
distance(D) :- constraint(distanceApart, N1, N2, P),
    D = #sum{X1: groupPos(N1,x,X1); Y1: groupPos(N1,y,Y1);
    Z1: groupPos(N1,z,Z1); -1*X2: groupPos(N2,x,X2); -1*Y2:
    groupPos(N2,y,Y2); -1*Z2: groupPos(N2,z,Z2)}.
distance(D) :- constraint(distanceApart, N2, N1, P),
```

```
    D = #sum{X1: groupPos(N1,x,X1); Y1: groupPos(N1,y,Y1);
    Z1: groupPos(N1,z,Z1); -1*X2: groupPos(N2,x,X2);
    -1*Y2: groupPos(N2,y,Y2); -1*Z2: groupPos(N2,z,Z2)} < D.

:- constraint(distanceApart, N2, N1, params(C)),
   distance(D), D < C.
```

Finally, we have the inside constraint — this forces a piece group to be completely contained within a volume group

```
{groupPos(N1,Axis,P2+1..P2+S2-S1-1)} = 1 :-
   finalConstraint(inside,N1,N2,P), groupPos(N2,Axis,P2),
   groupSize(N2,Axis,S2), groupSize(N1,Axis,S1),
   axis(Axis), P2 <= M, max(Axis,M), P2 > 0.
```

6.5.5 sketch.lp

Finally, we have sketch.lp, which has the core logic for determining the position and size of all of the individual Lego groups within the model. Here, we break out the size and position facts into three separate facts for each of the different axes. This is done for performance reasons – if we had left them as pos(X,Y,Z) facts, then each group under consideration would get grounded out with $X * Y * Z$ different position facts per group, versus having them as three separate facts which only has $X + Y + Z$ different facts per group. This has a significant impact on the system's performance when

```
{groupOrientation(N,Axis) : notUp(Axis)} = 1
   :- finalChosen(N,piece(P)).
groupOrientation(N,Axis)
```

```
   :- finalChosen(N,piece(P)), lockOrient(N,Axis).

groupSize(N,Axis,X) :- finalChosen(N,piece(P)),
   groupRef(X,Y,Z,P), groupOrientation(N,Axis).
groupSize(N,y,Y) :- finalChosen(N,piece(P)), groupRef(X,Y,Z,P).
groupSize(N,Other,Z) :- finalChosen(N,piece(P)),
   groupRef(X,Y,Z,P), groupOrientation(N,Axis),
   axisPair(Axis,Other).

{groupSize(N,x,X1..X2)} = 1
   :- finalChosen(N,volume(T,params(X1,X2,Y1,Y2,Z1,Z2))).
{groupSize(N,y,Y1..Y2)} = 1
   :- finalChosen(N,volume(T,params(X1,X2,Y1,Y2,Z1,Z2))).
{groupSize(N,z,Z1..Z2)} = 1
   :- finalChosen(N,volume(T,params(X1,X2,Y1,Y2,Z1,Z2))).

{groupPos(N,x,X) : xPos(X)} = 1 :- finalChosen(N,T).
{groupPos(N,y,Y) : yPos(Y)} = 1 :- finalChosen(N,T).
{groupPos(N,z,Z) : zPos(Z)} = 1 :- finalChosen(N,T).

:- 2 { groupPos(N,A,_) }, finalChosen(N,T), axis(A).
:- 2 { groupSize(N,A,_) }, finalChosen(N,T), axis(A).
```

Finally, we have a check to see if two piece groups intersect each other — we don't need to worry about volume groups since, because they're made out of lots of smaller pieces, we can just route around the embedded parts and other intersecting volume groups.

```
intersection(N1,N2,Axis) :- finalChosen(N1,piece(T1)),
   finalChosen(N2,piece(T2)), axis(Axis), N1 != N2,
   groupSize(N1,Axis,S1), groupSize(N2,Axis,S2),
   S1 -1 >= #sum{P2 : groupPos(N2,Axis,P2); -1*P1
      : groupPos(N1,Axis,P1)},
   1 - S2 <= #sum{P2 : groupPos(N2,Axis,P2); -1*P1
      : groupPos(N1,Axis,P1)}.

fullInter(N1,N2) :- intersection(N1,N2,x),
   intersection(N1,N2,y), intersection(N1,N2,z).

:- fullInter(N1,N2).
```

6.6 Realizer Implementation

Now, we move on to the Realizer. In this portion of the system, we take the sketches created by the Sketcher and transform them into full Lego models, placing all of the individual Lego pieces in place. This stage varies in complexity based on the kinds of groups in the model – for Lego models comprised of small volume groups and piece groups, this stage is nearly trivially. For models that feature large piece groups that need to be appropriately constructed, the process of this system becomes complex and requires a large amount of machinery to appropriately construct.

To that end, this portion of the system underwent development in 3 distinct stages of the core functionality. Before we discuss the different versions of the system, let's first look into the consist parts of this system, starting with the input.

The realizer takes its input from the sketcher as a list of ASP facts. Each of the Lego groups included in a sketch have the following classes of facts:

- finalChosen(name(N), piece(P)) – each group has one of these facts, as an identity label.
- groupPos(name(N),Axis,Pos) – as mentioned previously, breaking these facts up into 3 separate facts rather than a single position fact helps make ASP run more optimally, so we have one fact for each of the three coordinates
- groupSize(name(N),Axis,Pos) – identically to the above, but expressing the size of the group instead of the position.

We also hold on to a certain amount of additional information that we utilize in the final rendering process, but this information just gets passed out the other end of the pipeline without being affected. We hold on to the color information for the different groups as well as the orientation of the groups for the end.

With these, we then derive the cells that we need to fill in – specifically, determining which 1x1x1 cells within the space need to be filled in with pieces. For these, piece groups are treated as already filled in and finished. Volume groups, on the other hand, are reduced to volumes to be filled in. These facts come in two flavors – toFill(X,Y,Z) are locations where we want to fill the space, and bad(X,Y,Z) are locations that we don't want filled it. For instance, if we have hollow volume, we model this as toFill() facts that cover the entire volume, with bad() facts to represent the volume that we want to carve away instead.

Once we have the this space hammered out, we then need to fill it in with pieces. To do so, we use a set of pieces of width 1 and 2, and of lengths 1, 2, 3, 4, 6 and 8. Additionally, pieces come in two different heights — 1 (the height of a Lego plate) and 3 (the height of a Lego brick). These are listed as piece(X,Y) facts

within the system, and these are converted into piece(X,Y,Z) facts (representing the coordinates of their final locations within the space).

6.6.1 Naive Attempt

In our initial approach, we utilized a very simple heuristic solution process, utilizing a small set of constraints. For this, we specify a list of possible pieces—for the walls, this is a set of basic Lego bricks and plates, and we also include a set of sloped pieces for the roof—as a list of `legoCollection/3` facts, specifying the size of each piece. We also include our information from the sketch, transforming into a set of voxel coordinates that need to be filled as a list of `target/3` facts—including the space of each volume group, then subtracting out the space encapsulated by the piece groups.

- The target volume, and only the target volume, is covered with pieces.

```
pos((X,Y,Z),X,Y,Z) :- target(X,Y,Z).
{ blank(N);
  legoPiece(N, loc(I,J,K), size(X,Y,Z)) :
  legoCollection(X,Y,Z),
  target(X+I-1,Y+J-1,Z+K-1)} = 1
  :- pos(N,I,J,K).
coveredBy(N,I1,J1,K1) :-
  legoPiece(N, loc(I,J,K), size(X,Y,Z)),
  I1 = I..I+X-1,
  J1 = J..J+Y-1,
  K1 = K..K+Z-1.
covered(X,Y,Z) :- coveredBy(N,X,Y,Z).
```

```
:- target(X,Y,Z), not covered(X,Y,Z).
:- covered(X,Y,Z), not target(X,Y,Z).
```

- No two pieces used intersect with each other

```
:- pos(N,X,Y,Z),
  2 { coveredBy(Ni,X,Y,Z) }.
```

- No two identical pieces are directly on top of each other (decreasing the potential for free-standing sections of the model)

```
:- legoPiece(N1, loc(I,J,K), size(X,Y,Z)),
 legoPiece(N2, loc(I,J+Y,K), size(X,Y,Z)).
```

- The minimum number of Lego pieces are used in the final model.

```
#minimize {1@1,N,L,S: legoPiece(N,L,S)}.
```

This was theoretically sound, and produced models as big as the houses we saw in the previous chapter. However, this approach had a number of distinct limitations. In clingo, finding the smallest number of pieces works by exhaustively proving that there are no solutions with a smaller number of pieces, each of which could be at any location within the 3 dimensional volume. So, for a 10-by-10-by-10 volume, if we want to disprove that we can cover it with 20 pieces, we need to examine 10^{60} different piece layouts. Luckily, in practice, the underlying algorithms that clingo provides help us ignore a lot of the redundant combinations, so the actual number of combinations we wish to look at is likely several orders of magnitude smaller; however, being orders of magnitude smaller than 10^{60} is still very large. Additionally, as we grow the size of the model, the number of pieces

required to cover the model trends towards increasing, so even small increases beyond the size of the house model let to the system taking extraordinary amounts of time to reach a solution.

We attempted to compensate by making updates to the underlying solver, tightening things up; adding in additional constraints on where pieces could be places, etc. However, all of these modifications were dancing around the previous underlying problem of exponential growth.

6.6.2 Layers

In order to combat the explosive problem-size growth we saw, it was necessarily to reformulate the problem to be more manageable. As we know from the Sketcher, one way to reduce complexity in ASP is to break choices of variables up into separate pieces; if we solve for three loc(Value,Axis) rules, rather than one loc(Xvalue, Yvalue, Zvalue) rule, the total number of possible facts is $X + Y + Z$ rather than $X * Y * Z$. So, we turned to real Lego models for inspiration.

In actual Lego models, where we have large standard-brick structures (what we would call volume groups), we see that pieces reuse the same patterns to build up the walls – if no other pieces interrupt the horizontal slices, there's no reason to not alternate between two different patterns to fill in the repeated space.

This was a useful find, allowing us to do two major things:

1. Reduce the 3D problem to a set of 2D problems, which greatly reduces the search space for any given piece.

2. Reduces the effective size of the vertical axis by minimizing the number of distinct layers, and reusing solutions across identical layers.

To do this, we reformulated the problem as follows:

- We start by creating a list of row facts that describe all of the possible Y-values for pieces. From there, we build a set of layers, of height either 1 or 3, to represent that layer being either plate-height or brick-height.

```
row (Y) :- good(X,Y,Z).
{layerPos(X) : row(X)}.
%pieceY is either 1 or 3.
{layer(X,Y) : pieceY(Y)} = 1 :- layerPos(X).
covered(X..X+I-1) :- layer(X,I).
:- row(Y), not covered(Y).
:- covered(Y), not row(Y).
```

- With those in place, we then add some checking to make sure that the layers get grouped with other identically shaped layers. To do this, we first identify rows that feature some kind of difference (here defined as a different in cells that need to be filled in), and then build a series of facts describing the rows that have no differences. Additionally, we can't have a brick-sized layer if there's any differences between the piece-sized layers that make it up.

```
rowMismatch(A,B) :- row(A), row(B),
   good(X,A,Z), not good(X,B,Z).
rowMismatch(A,B) :- row(A), row(B),
   bad(X,A,Z), not bad(X,B,Z).
rowMismatch(A,B) :- row(A), row(B),
   occupied(X,A,Z), not occupied(X,B,Z).
rowMismatch(A,B) :- rowMismatch(B,A).
rowMatch(A,B) :- row(A), row(B), not rowMismatch(A,B), A != B.
:- layer(X,3), rowMismatch(X,X+1..X+2).
```

- Then, we want to reduce the layers to a single, unique instance of each set of identical layers; we call these the layerExemplars. Here, we pick one Exemplar for each set of rows, ensuring that we only have one per set and that by their powers combined, they represent the entire space of the Lego model.

```
{layerExemplar(X) : layer(X,I)}.
:- layerExemplar(X), layerExemplar(Y), rowMatch(X,Y).
layerCover(X) :- layerExemplar(X).
layerCover(X) :- rowMatch(X,Y), layerCover(Y).
:- row(X), not layerCover(X).
```

From here, we can solve for a layout that covers each of the individual layer exemplars, and we technically have a solution for the overall model. However, this approach does not necessarily support one of the fundamental constraints we've used for Lego models - namely, that identical pieces shouldn't be stacked on top of each other. As it stands, if we only solve for one solution per layer, we will break this constraint if we have two identical layers stacked on top of each other. So, we add one additional piece of machinery – we solve for two distinct solutions per layer (call them solutions 1 and 2) with the constraint that each solution 1 needs to stack well against each solution 2 – or, at least, as well as possible.

```
exemplarIDs(1..Y) :- exemplarCount(Y).
{layerBijection(X,Y) : layerExemplar(X)} = 1 :- exemplarIDs(Y).
{layerBijection(X,Y) : exemplarIDs(Y)} = 1 :- layerExemplar(X).

layersToBuild(1..Y,1..2) :- exemplarCount(Y).
toTile(pos(X,Z),layer(Y,ID)) :- layersToBuild(Y,ID),
```

```
   good(X,J,Z), layerBijection(J,Y).
{ root(Pos,Layer) : toTile(Pos,Layer)}.
{ piece(Pos,(I,J),Layer) : piece(I,J) } = 1 :- root(Pos,Layer).
```

Finally, we do some last minute validation – we include the constraints to check that the volume with want to fill – and only that volume – is filled in with pieces. We also include the directives to minimize the number of pieces and the number of identical pieces stacked on top of each other.

```
covered(pos(X..X+I-1,Z..Z+J-1),Layer)
   :- piece(pos(X,Z),(I,J),Layer).
:- covered(Pos,Layer), not toTile(Pos,Layer).
:- not covered(Pos,Layer), toTile(Pos,Layer).


stacked(Pos,L1,L2) :- piece(Pos,Size,(L1,1)),
   piece(Pos,Size,(L2,2)).


#minimize {1@5,Pos,Size,Layer: piece(Pos,Size,Layer)}.
#minimize {1@1,Pos,L1,L2 : stacked(Pos,L1,L2)}.
```

This greatly helped with our solving performance, however even this wasn't enough – we still were getting bogged down in the solution. However, at this point, there wasn't much left we could do within the realm of pure Answer Set Programming.

6.6.3 Layers with Propagation Control

On of the lesser used features in clingo is the python api, which provides access to the underlying solver's functionality. The final bit of functionality that we add

to the solver, in order to gain the maximum performance we can, is to override the default solver functions for the solver, forcing them to introduce extra constraints on particular choices. In the init() function, we tell the system to watch the layer(X,Y) facts – our custom propagate() function puts constraints on which other layer(X,Y) facts can exist in the same answer set. In this way, we ensure that we minimize the amount of backtracking we need to do with regard to the layers.

```
#script (python)
import clingo
class Propagator:

    def __init__(self):
        self.layers = {}
        self.layerPos = {}
        self.constraintList = []
        return

    def init(self, init):
        for atom in init.symbolic_atoms.by_signature("layer",2):
            lit = init.solver_literal(atom.literal)
            self.layers[lit] = \
               (int(atom.symbol.arguments[0].number), \
                int(atom.symbol.arguments[1].number))
            if lit != 1L:
                init.add_watch(lit)
        for atom in init.symbolic_atoms.by_signature("layerPos",1):
```

```
            lit = init.solver_literal(atom.literal)
            self.layerPos[int(atom.symbol.arguments[0].number)] \
               = lit
        return

    def propagate(self, control, changes):
        for lit in changes:
            layer = self.layers[lit]
            #(Pos,Size)
            for i in range(1,layer[1]):
                self.constraintList.append([lit, \
                self.layerPos[layer[0]+i]])
        constraintControl = len(self.constraintList) > 0
        while constraintControl:
            adding = self.constraintList.pop()
            constraintControl = \
                control.add_nogood(adding,False,True) \
                and len(self.constraintList) > 0

        return

    def undo(self, solver_id, assign, undo):
        return

    def check(self, control):
        constraintControl = len(self.constraintList) > 0
```

```
        while constraintControl:
            adding = self.constraintList.pop()
            constraintControl = \
                control.add_nogood(adding,False,True) \
                and len(self.constraintList) > 0
        return



#Boilerplate
def main(prg):
    prg.register_propagator(Propagator())
    prg.ground([("base",[])])
    prg.solve()


#end.
```

As we can see in Figure 6.1, this new system is able to produce much larger models that the basic houses we saw previously.

6.7 Conclusion

In this chapter, we discussed the Answer Set Programming implementation of the Solus Forge system. By implementing this system using ASP, we were able to demonstrate the power and limitations of the hierarchical decomposition framework – namely, while the framework allows us a way to combine arbitrary parts of a model together, we need an extremely powerful engine to ensure that we can generate things fully. Additionally, we ran into some of the limitations of

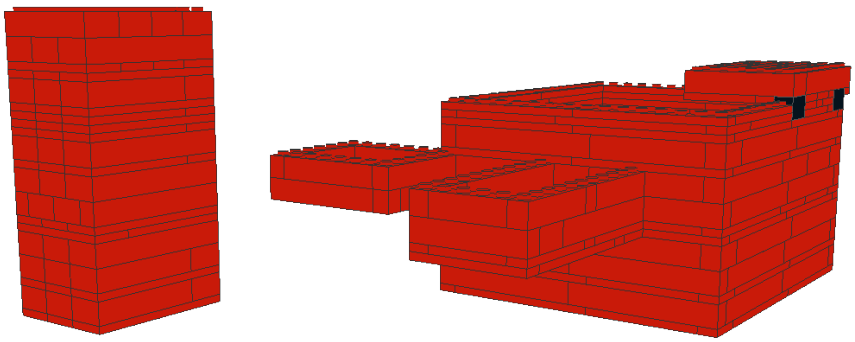

Figure 6.1: A model, demonstrating a very large model constructed by the final realizer.

clingo during the project, some of which were the result of our domain of Legos. In the end, however, we were able to utilize some of the lesser-used Clingo features in order to overcome the performance issues we faced.

Chapter 7

Evaluation

> *In theory, there's no difference between theory and practice. In practice, there is.*
>
> — Unknown

7.1 Introduction

Given my implementation of Solus Forge, the immediate question that comes to mind is "How impactful are these stylistic transformations, really?" We can see the immediate results in individual artifacts, but to test if these transformations reach the impact we desire, we need to see how they affect not just individual artifacts, but the entire generative space.

In this chapter, we preform an expressive range analysis on Solus Forge in order to show how different stylizations affect the overall output of the generator.

7.2 Expressive Range

Expressive Range Analysis was a term coined by Smith, Whitehead and Mateas to provide a tool for understanding the space of a generator, specifically for the

Tanagra system (a platformer level generator) [88]. To quote the paper:

> While it is easy to show that Tanagra can create a large quantity of levels, we feel it is more interesting to examine the qualities of the levels that are produced, and compare how similar they are to each other.

To this end, the authors created two metrics for the different levels that a generator could produce, and plotted this metrics for a sampling of levels created by a given generator. While the original paper only considers the space examined by a single generator, subsequent papers [94] have since used these plots to compare different generators that operate over the same space. In this way, we can examine artifacts fall within the theoretical total possible space of artifacts to see which areas a given generator favors or disfavors.

The metrics choosen for Tanagra were *linearity*, a measure of how flat the terrain of the level was, and *leniency*, which measured the number of obstacles the player faced in the level, providing a measure that made gestures towards being a measure of difficulty. While these metrics make sense for a 2D platformer, they are by no means the only ones that could have been chosen, even within that domain in particular. In [94], Summerville looks at Mario levels through the lens of no fewer than eight distinct metrics to compare the different generators examined in the work.

For the purposes of Lego models, we need to construct other metrics that we can utilize to convey how different stylizations affect the generative space. The simplest metric to consider is how the style affects the shape of the output model, since most of the style rules affect which pieces are (or are not) included in the final output model. For this, we could look at the total number of pieces used, the volume that the model takes up, the size of the bounding box around the model, or any number of other metrics. After consideration, we chose positional variance

as our metric. We define this metric as follows: for each piece n, we compute its positional variance pv_n along each of the 3 axes as follows:

$$pv_n = (\mu - p_n)^2$$

where μ is the mean position of all the pieces along the given axis, and p_n is the position of the piece along the same axis. The positional variance of the entire model is defined as the mean of all of the positional variances across every piece and axis.

The main benefit of this metric is that small changes in the layout of a model produce noticeable changes in the outcome of the metric. For instance: assume that the mean position of a model is at the origin $(0, 0, 0)$, and it is made of two pieces, one at $(5, 5, 5)$ and one at $(-5, -5, -5)$. The positional variance of this model is

$$((0-5)^2 + (0-5)^2 + (0-5)^2 + (0+5)^2 + (0+5)^2 + (0+5)^2)/6 = 25$$

If we move the first piece to $(6, 5, 5)$ instead, the mean position becomes $(.5, 0, 0)$, and the positional variance becomes:

$$((.5-6)^2 + (0-5)^2 + (0-5)^2 + (.5+5)^2 + (0+5)^2 + (0+5)^2)/6 = 26.75$$

which is a noticeable change. These differences compound as the number of pieces in the model increase, since at most one piece can be at the center.

For our other metric, we consider average hue, here defined as the volumetric average of the hues of each piece of the model. For each 1-by-1-by-1 segment of the model, we take the hue of the piece at that location, and compute the average over the total volume. Of course, since hue is a modular space (the difference

between hue 320 and hue 20 is 60, not 300), doing a simple sum does not suffice here, but there is a rather useful out. Since hue values are given in the range $[0, 360)$, we can view these as angles along a circle, and hence to compute the mean σ of hues $h_1, h_2, ...h_n$, we do:

$$\sigma = \arctan(\sum_{i=1...n} (\cos(h_i))/n, \sum_{i=1...n} (\sin(h_i))/n)$$

For instance, if we have a model composed of a green 2x3x4 brick (hue value 138) and an orange 1x3x2 brick (hue value 30), the average hue over the total volume of 30 units is:

$$\arctan((6\cos(30) + 24\cos(138))/30, (6\sin(30) + 24\sin(138))/30) = 123.55$$

This metric allows us to determine subtle changes in coloration in a model — from the addition or remove of a piece, from a piece's color being changed, or even from a piece being resized (since a resized piece would provide a different volume of the given color to the new model). This final property allows us to cover a major weakness of the previous metric: the positional variance value won't change if a piece is resized if it doesn't impact any other pieces.

While these metric was derived specifically for these Lego models, one can imagine other sorts of domains utilizing the metric in order to differentiate between differences in form. The positional variance gives an insight into how spread out the different parts of the model are from the center, and can be treated as a variant on Tanagra's linearity metric.

With these metrics in place, we're in a good place to start examining how different the styles that our generator can produce. To do this, I created a small

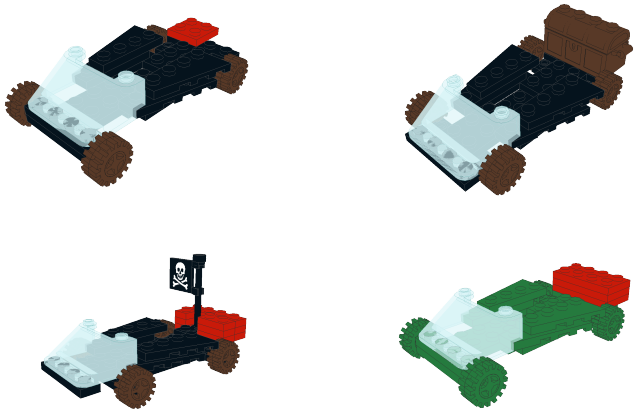

Figure 7.1: Example model outputs for the Standard, AddChest, AddFlag and Green styles

group of sample styles to show how the generator transforms its expressive range in response to the stylistic changes. These include:

- AddChest: Adds a single Treasure chest to the Lego Model
- AddFlag: Adds a single Flag to the Lego Model
- GreenHue: Colors most of the model green
- ChestPlusColor: Applies both GreenHue and AddChest style rules

These cover a range of test cases to demonstrate how the styles transform both the individual models, and the space itself. In Figure 7.1, we can see an example model from each of the styles we're examining in this space. The different styles each feature different total numbers of models – the standard model is fairly constrained, but the AddChest style features a large number of possibilities due to the system deciding where the added chest should be placed. We can see the total numbers in Table 7.1.

Style Name	Model Count
Standard	54
AddChest	13716
AddFlag	14364
GreenHue	54
ChestPlusColor	13716

Table 7.1: Table of styles and their various total model counts

7.3 Testing Setup

To test these, we utilize several features of Answer Set Programming to ensure full coverage of our test cases. ASP allows us to set the number of output models we get from any particular run of the system; while tuning parameters to ensure a representative sample can be tricky, we sidestep this by setting the number of output models to 0, which ASP interprets as "return all of the solutions." In this way, we do not need to account for sampling bias, since our sample is just the entire pool. We use this to generate all possible sketches, which form our set of outputs from the stylized model.

With those, we compute both the average hue and positional variance of each individual Lego model. In this way, we can see our results from both the breadth of the space as well as the results of the generated models themselves.

7.4 Results and Analysis

We start by showing the results that we generated from the initial runs—if we look at Figure 7.2, we see results from 4 of our different stylizations: first the default car model, followed by the AddChest style, the Green style and finally the ChestPlusColor style.

Here, we can clearly see a few obvious results from the Positional Variance: the Standard and Green styles both shared identical spreads across all of the

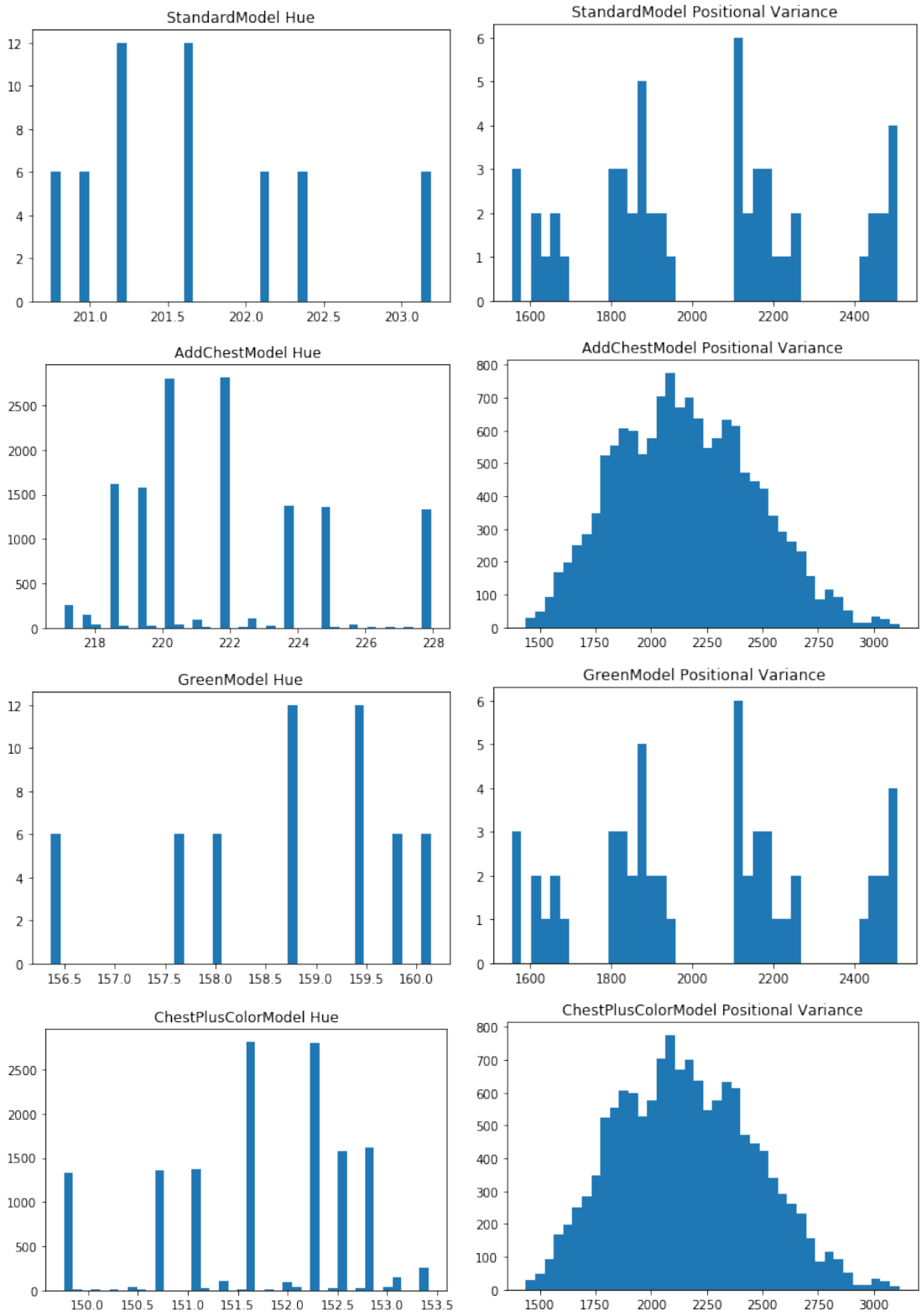

Figure 7.2: Our Expressive Range Data - the charts are arranged in four rows, each containing the data from one particular style. The top row is the Standard style, followed by AddChest, Green and lastly ChestPlusColor.

models; and the AddChest and ChestPlusColor styles featured the same property, while being significantly different from the other two. This follows because the difference between the matching pairs is only the color of the different bricks, while the mismatched pairs show the results of including the treasure chest piece in the model.

In contrast the Hue charts are a little less obvious, since the treasure chest is brown, a hue that differs from the main hues of the Standard model - and so we see a shift in the hue there. However, we do see a stark difference in hue range between those models and the Green and ChestPlusColor models, both of which have their average hues around 50 units lower, in the 150 range (pure green is hue 120). In summary, we can see the dramatic effect that these style rules have on the models' expressive ranges.

In Figure 7.3, we see a comparison between the two different styles that each add a group to the model, the ChestAdd and the FlagAdd styles. Here, we can see that the two styles both affect the expressive range in similar, but distinct ways — the overall shapes of the histograms is the same for both styles, but the axes are slightly different, especially for Average Hue. FlagAdd's Average Hue is very similar to the Standard Model, while the extra brown volume from the treasure chest pushes AddChest up almost 20 units. This is further emphasized in Figure 7.7, where we plot the space of each model in both dimensions. Here, we can more clearly see the sparsity of the Standard Model compared with the others, as well as the massive hue shift we see from including the chest.

If we look at Figure 7.4 and 7.5, we can see how these play out in the extremes: in Figure 7.4, we see the minimum color values for each of the styles, and in Figure 7.5 we see the maximum values. Notice that, in each case, the primary change is the amount of red in each color — and in the AddChest style, we can see the

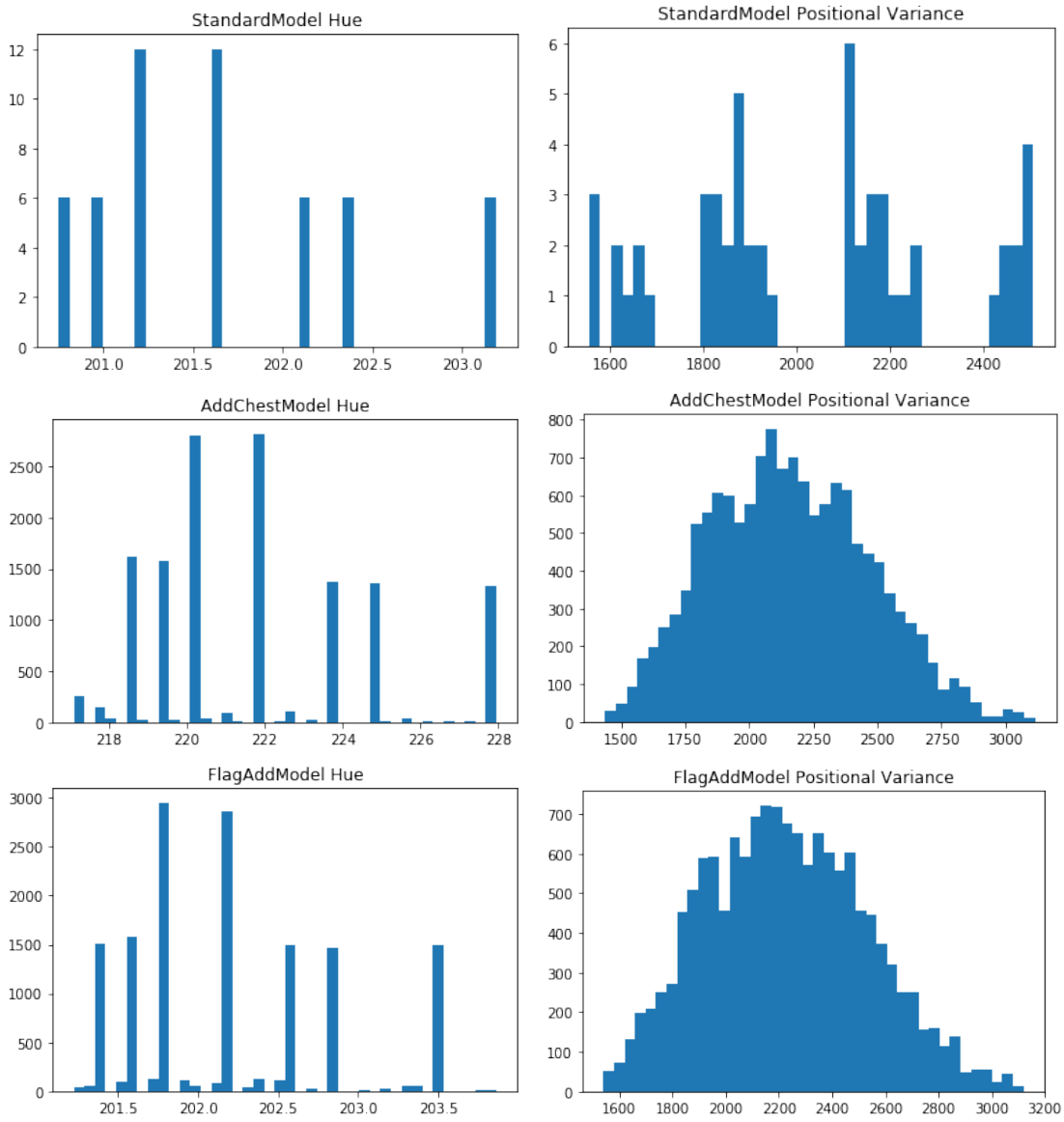

Figure 7.3: Our Expressive Range Data, Part 2; the Standard and AddChest data is repeated, but we also include the AddFlag style as well.

Figure 7.4: Examples of the minimum color value models for the Standard and AddChest styles

Red bricks completely vanish, compared to the Standard style's minimization — this, combined with the sheer amount of color data provided by the brown chest, explain the much larger difference in values between the minimum and maximum values. In contrast, adding a flag provides a much smaller difference in the color values; because the black of the flag adds a neutral amount to the mean color.

Figure 7.7 shows the results from plotting the 2D expressive range of the Standard, AddFlag and AddChest styles against each other - mostly reinforcing what we see in the earlier diagrams. It is interesting to note the stripes of average hue values (corresponding mostly to certain amounts of the red back of the model existing at various stages of the process), and how the differing stripes expand and contract along values of positional variance – signifying that different hue values naturally constrain the number of ways that the model can place the additional Lego groups.

As we can see, both the chest and the Flag Add styles provide a large spread of positional variance values – given that both of the models feature new pieces that can be added almost anywhere to the existing model, this large amount of positional variance is to be expected as the positions range from close to the center of the model to the furthest extremities.

Figure 7.5: Examples of the maximum color value models for the Standard and AddChest styles

Figure 7.6: Examples of the FlagAdd style

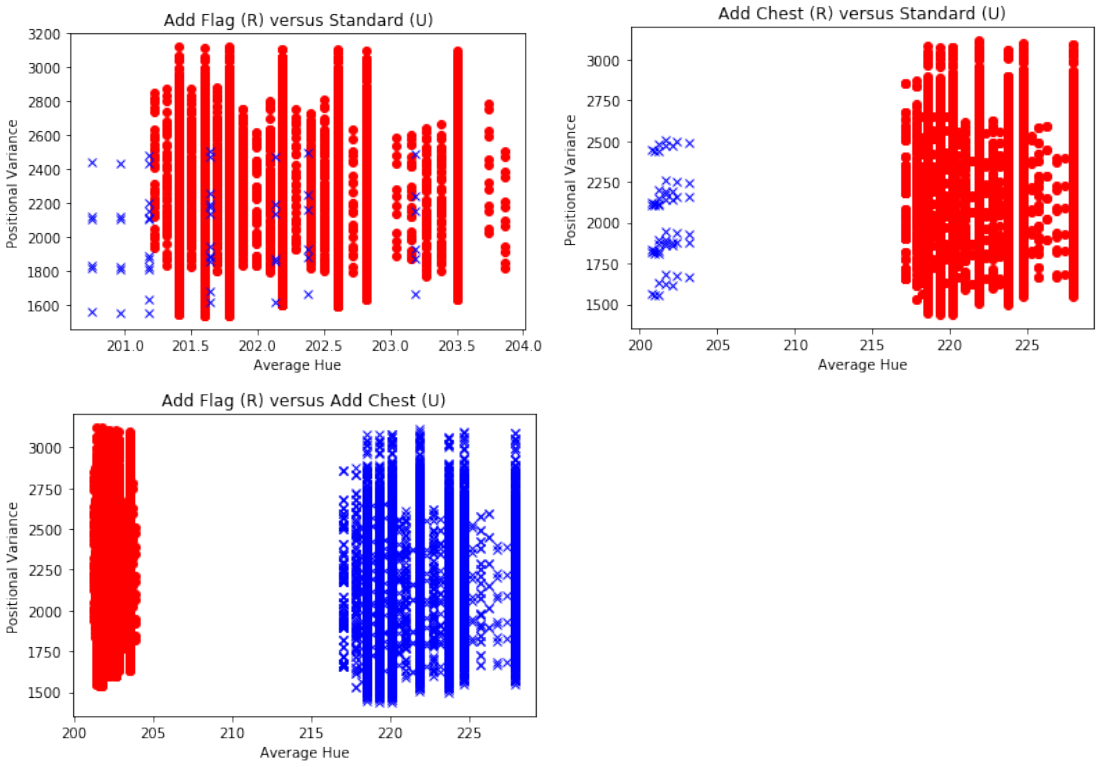

Figure 7.7: 2D plots of the expressive ranges, comparing the AddFlag, AddChest and Standard styles. In particular, each marker represents a single generated model in the given style, placed at the appropriate location for its positional variance and average hue values.

7.5 Conclusions

In conclusion, we can see that our style rules clearly impact the expressive range of our car model, with most of the impact coming from the specific style rules we choose. In this way, the style rules allow us to transform the basic generation process so that it expresses these new areas of the space, allowing us to control and guide the outcomes we desire.

Chapter 8

Conclusion

8.1 Summary

In this work, we provided several key contributions to the field of procedural content generation, and computer science more broadly. Firstly, we introduced the Hierarchical Decomposition framework for understanding, modeling and generating 3D models. In this model, we construct models as both a hierarchy of parts as well as a series of positional constraints between those part, providing both a guideline for how the models are constructed, but also flexibility in the space should the constraints be under-specified. We also provided a through survey and comparison of procedural systems that featured a focus on style, viewing them through the lens of the underlying technical processes that drive the work (from Grammars to multi-agent systems, to genetic algorithms and neural networks).

From these broad strokes, we implemented two novel systems for style-based PCG; specifically, two systems that utilized different forms of constraint solving as the core generative process. One, an unnamed system built on top of the semantic framework Rensa, used an implementation of the Rete algorithm to apply style rules to an abstract representation of a narrative. The other, Solus Forge, is a

system for generating stylized Lego models using Answer Set Programming to both do the stylization process, as well as the model generation.

8.2 Future Work

Moving forward, this work provides a number of foundations to be build upon. First and foremost, we provide the first known survey of style-focused procedural generation systems, which is important for contextualizing the works as being related. However, it necessarily focuses on visual works, which is important given the domain of this dissertation. Looking at works in other domains through this lens would be an important next step, as would looking at other cross-section of works presented.

Secondly, this work features two very different constraint-centric systems for generating stylized artifacts – these are only two of a number of different constraint frameworks that exist, each with their own sets of issues and features. Exploring stylization under other core engines is another rich avenue of work to explore, as well as continuing to explore different techniques within the frameworks we already have.

Finally, as we mention in other chapters, the procedural generation of 3D artifacts is a domain that is understudied within PCG literature. While Solus Forge only operates on a very particular and limited space within the domain of general 3D models – namely, Lego models – the template that it provides for utilizing the Hierarchical Decomposition framework to operate over 3D models needs to be expanded upon and extended to the less discrete domain of general 3D models. Part of this transformation would be to explore other underlying systems for generation (as Answer Set Programming has no floating-point support natively).

8.3 Final Thoughts

Style is an integral part of art and human expression – the design choices that inform the creative process are inescapable and so understanding how these influence an artifact's creation is critical to emulating that process within a computational system. Here, we're provided the groundwork for encoding and understanding style, in the context of 3D models — an underexplored domain for procedural generation as well.

In summary, this dissertation lays the groundwork for a new understanding of both style and 3D models in procedural generation.

Bibliography

[1] Out of context D&D quotes. `http://outofcontextdnd.tumblr.com/post/145623705582/no-no-no-dwarves-are-art-deco-elves-are-art`. Accessed: 2019-02-14.

[2] Procjam. `http://www.procjam.com/`. Accessed: 2019-02-14.

[3] Manish Agarwal and Jonathan Cagan. A blend of different tastes: The language of coffeemakers. *Environment and Planning B: Planning and Design*, 25(2):205–226, 1998.

[4] Taisuke Akimoto and Takashi Ogata. A narratological approach for narrative discourse: Implementation and evaluation of the system based on genette and jauss. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 34, 2012.

[5] Evgenia Antonova. Applying Answer Set Programming in game level design. Master's thesis, Aalto University, Espoo, Finland, Dec 2015.

[6] D.L. Ashliman. Grimm brothers' home page. `http://www.pitt.edu/~dash/grimm.html`. Accessed: 2018-06-02.

[7] Tarek R Besold and Enric Plaza. Generalize and blend: Concept blending based on generalization, analogy, and amalgams. In *Proceedings of the Sixth International Conference on Computational Creativity June*, page 150, 2015.

[8] Margaret Boden. *The Creative Mind*. Abacus, London, 1992.

[9] Georg Boenn, Martin Brain, Marina De Vos, and John Ffitch. Automatic Music Composition Using Answer Set Programming. *Theory and practice of logic programming*, 11(2-3):397–427, 2011.

[10] Grady Booch. *The Unified Modeling Language user guide*. Addison-Wesley Professional, 2nd edition, 2005.

[11] F Bou, M Schorlemmer, J Corneli, D Gómez-Ramírez, E Maclean, A Smaill, and A Pease. The role of blending in mathematical invention. In *Proceedings*

of the Sixth International Conference on Computational Creativity June, page 55, 2015.

[12] Joseph Campbell. *The hero with a thousand faces*, volume 17. New World Library, 2008.

[13] Alex J Champandard. Semantic style transfer and turning two-bit doodles into fine artworks. *arXiv preprint arXiv:1603.01768*, 2016.

[14] Kate Compton, Ben Kybartas, and Michael Mateas. Tracery: An author-focused generative text tool. In *International Conference on Interactive Digital Storytelling*, pages 154–161. Springer, 2015.

[15] Kate Compton, Joseph C Osborn, and Michael Mateas. Generative methods. In *The Fourth Procedural Content Generation in Games workshop, PCG 2013*, 2013.

[16] Thomas Crick. *Superoptimisation: Provably optimal code generation using Answer Set Programming.* PhD thesis, University of Bath, 2009.

[17] Armando de la Re, Francisco Abad, Emilio Camahort, and M Carmen Juan. Tools for Procedural Generation of Plants in Virtual Scenes. In *International Conference on Computational Science*, pages 801–810. Springer, 2009.

[18] Andrés Gómez de Silva Garza and Aram Zamora Lores. Automating evolutionary art in the style of Mondrian. In *Genetic and Evolutionary Computation Conference*, pages 394–395. Springer, 2004.

[19] Marta Destler. Elves and art nouveau. `https://middleeartharchitectures.com/2014/12/21/elves-and-art-nouveau/`. Accessed: 2019-02-21.

[20] Marta Destler. Something about dwarven architectures. `https://middleeartharchitectures.com/2019/02/04/something-about-dwarven-massive-architectures/`. Accessed: 2019-02-21.

[21] Alexandre Devert, Nicolas Bredeche, and Marc Schoenauer. Blindbuilder: A New Encoding to Evolve Lego-like Structures. In *European Conference on Genetic Programming*, pages 61–72. Springer, 2006.

[22] Carl Doersch, Saurabh Singh, Abhinav Gupta, Josef Sivic, and Alexei Efros. What makes Paris look like Paris? *ACM Transactions on Graphics*, 31(4), 2012.

[23] Jonathon Doran and Ian Parberry. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):111–119, 2010.

[24] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer Set Programming: A Primer. In *Reasoning Web. Semantic Technologies for Information Systems*, pages 40–110. Springer, 2009.

[25] David K Elson. Detecting story analogies from annotations of time, action and agency. In *Proceedings of the LREC 2012 Workshop on Computational Models of Narrative, Istanbul, Turkey*, 2012.

[26] Manfred Eppe, Ewen Maclean, Roberto Confalonieri, Oliver Kutz, Marco Schorlemmer, and Enric Plaza. ASP, amalgamation, and the conceptual blending workflow. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 309–316. Springer, 2015.

[27] Gilles Fauconnier and Mark Turner. Conceptual integration networks. *Cognitive science*, 22(2):133–187, 1998.

[28] Margaret Finch. *Style in Art History*. Metuchen, N.J., Scarecrow Press, 1974.

[29] Margaret Finch. Style in art history an introduction to theories of style and sequence. 1974.

[30] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Readings in Artificial Intelligence and Databases*, pages 547–559. Elsevier, 1989.

[31] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*, 2015.

[32] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + Control: Preliminary Report. In Leuschel and Schrijvers [55]. Theory and Practice of Logic Programming, Online Supplement.

[33] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The Potsdam answer set solving collection. *Ai Communications*, 24(2):107–124, 2011.

[34] Joseph A Goguen and D Fox Harrell. Style: A computational and conceptual blending-based approach. In *The Structure of Style*, pages 291–316. Springer, 2010.

[35] Dorrit H Gordon and E James Whitehead. Containment modeling of content management systems. In *International Symposium on Metainformatics*, pages 76–89. Springer, 2002.

[36] Markus Guhe, Alison Pease, Alan Smaill, Maricarmen Martinez, Martin Schmidt, Helmar Gust, Kai-Uwe Kühnberger, and Ulf Krumnack. A computational account of conceptual blending in basic mathematics. *Cognitive Systems Research*, 12(3):249–265, 2011.

[37] Matthew Guzdial and Mark Riedl. Learning to blend computer game levels. In *Proceedings of the Seventh International Conference on Computational Creativity June*, pages 354–362, 2016.

[38] Gaëtan Hadjeres, François Pachet, and Frank Nielsen. Deepbach: A steerable model for bach chorales generation. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1362–1371. JMLR. org, 2017.

[39] Sarah Harmon. An expressive dilemma generation model for players and artificial agents. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[40] Mark Hendrikx, Sebastiaan Meijer, Joeri van der Velden, and Alexandru Iosup. Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 9(1):Article 1, February 2013.

[41] Douglas Hofstadter and Gary McGraw. Letter spirit: An emergent model of the perception and creation of alphabetic style. In *Technical Report 68, Center for Research on Concepts and Cognition*. 1993.

[42] Douglas Hofstadter and Gary McGraw. Letter spirit: Esthetic perception and creative play in the rich microcosm of the roman alphabet. In *Fluid concepts and creative analogies*, pages 407–466. Basic Books, Inc., 1995.

[43] Douglas R Hofstadter. *Fluid concepts and creative analogies: Computer models of the fundamental mechanisms of thought.* Basic books, 1995.

[44] Shohei Imabuchi and Takashi Ogata. Methods for generalizing the propp-based story generation mechanism. In *International Conference on Active Media Technology*, pages 333–344. Springer, 2013.

[45] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *arXiv preprint arXiv:1611.07004*, 2016.

[46] J Jessiman. LDraw, LEGO CAD software package, 1995. Accessed: 2019-03-24.

[47] Yongcheng Jing, Yezhou Yang, Zunlei Feng, Jingwen Ye, Yizhou Yu, and Mingli Song. Neural style transfer: A review. *arXiv preprint arXiv:1705.04058*, 2017.

[48] Maximos Kaliakatsos-Papakostas, Roberto Confalonieri, Joseph Cornelic, Asterios Zacharakisa, and Emilios Cambouropoulosa. An argument-based creative assistant for harmonic blending. In *Proceedings of the Seventh International Conference on Computational Creativity June*, pages 354–362, 2016.

[49] Darius Kazemi. Spelunky generator lessons. `http://tinysubversions.com/spelunkyGen/`, 2013. Accessed: 2019-03-17.

[50] Hank Koning and Julie Eizenberg. The language of the prairie: Frank Lloyd Wright's prairie houses. *Environment and Planning B: Planning and Design*, 8(3):295–323, 1981.

[51] Kathrin Koslicki. *The structure of objects*. Oxford University Press on Demand, 2008.

[52] Becky Lavender and Tommy Thompson. The Zelda Dungeon Generator: Adopting Generative Grammars to Create Levels for Action-Adventure Games, April 2015. Undergraduate Honors Thesis.

[53] HC Lee and MX Tang. Evolutionary Shape Grammars for Product Design. *Proceedings of Generative Art 2004*, 2004.

[54] Ho Cheong Lee and Ming Xi Tang. Evolving product form designs using parametric shape grammars integrated with genetic programming. *AI EDAM*, 23(2):131–158, 2009.

[55] M. Leuschel and T. Schrijvers, editors. *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, volume 14(4-5), 2014. Theory and Practice of Logic Programming, Online Supplement.

[56] Håkon Wium Lie and Bert Bos. *Cascading style sheets: Designing for the Web*. Addison-Wesley, 3rd edition, 2005.

[57] Vladimir Lifschitz. What is Answer Set Programming? In *AAAI*, volume 8, pages 1594–1597, 2008.

[58] Benjamin Mark, Tudor Berechet, Tobias Mahlmann, and Julian Togelius. Procedural Generation of 3D Caves for Games on the GPU. In *FDG*, 2015.

[59] Jay P McCormack, Jonathan Cagan, and Craig M Vogel. Speaking the Buick language: Capturing, understanding, and exploring brand identity with shape grammars. *Design studies*, 25(1):1–29, 2004.

[60] Benachir Medjdoub and Bernard Yannou. Dynamic Space Ordering at a Topological Level in Space Planning. *Artificial Intelligence in Engineering*, 15(1):47–60, 2001.

[61] Paul Merrell. Example-based Model Synthesis. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pages 105–112. ACM, 2007.

[62] Leonard B. Meyer. Toward a theory of style. In Leonard B. Meyer and Berel Lang, editors, *The Concept of Style*, pages 3–44. University of Pennsylvania Press, 1979.

[63] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, 1996.

[64] Yuki Mori and Takeo Igarashi. Plushie: An Interactive Design System for Plush Toys. In *ACM Transactions on Graphics (TOG)*, volume 26, page 45. ACM, 2007.

[65] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural Modeling of Buildings. In *ACM Transactions On Graphics (TOG)*, volume 25, pages 614–623. ACM, 2006.

[66] David R Nadeau. Building virtual worlds with VRML. *IEEE Computer Graphics and Applications*, 19(2):18–29, 1999.

[67] Mark J Nelson and Adam M Smith. ASP with applications to mazes and levels. In *Procedural Content Generation in Games*, pages 143–157. Springer, 2016.

[68] L Neumann, M Sbert, B Gooch, W Purgathofer, et al. Defining computational aesthetics. *Computational aesthetics in graphics, visualization and imaging*, pages 13–18, 2005.

[69] Takashi Ogata. Building conceptual dictionaries for an integrated narrative generation system. *Journal of Robotics, Networking and Artificial Life*, 1(4):270–284, 2015.

[70] Santiago Ontanon and Jichen Zhu. The SAM algorithm for analogy-based story generation. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[71] Seth Orsborn, Jonathan Cagan, Richard Pawlicki, and Randall C Smith. Creating cross-over vehicles: Defining and combining vehicle classes using shape grammars. *AI EDAM*, 20(3):217–246, 2006.

[72] Johnathan Pagnutti, Kate Compton, and Jim Whitehead. Do you like this art I made you: Introducing Techne, a creative artbot commune. In *Proceedings of 1st International Joint Conference of DiGRA and FDG*, 2016.

[73] Yanxin Pan, Alex Burnap, Ye Liu, Honglak Lee, Richard Gonzalez, and Panos Papalambros. A quantitative model for identifying regions of design visual attraction and application to automobile styling. In *Proceedings of the 2016 Internation Design Conference*, 2016.

[74] Peggy Pardo. Decorating style: The difference between art deco and art nouveau. `https://thecasacollective.com/art-deco-and-art-nouveau/`. Accessed: 2019-02-14.

[75] Yoav IH Parish and Pascal Müller. Procedural Modeling of Cities. In *Proceedings of SIGGRAPH 2001*, pages 301–308. ACM, 2001.

[76] Pavel Petrovic. Solving LEGO Brick Layout Problem Using Evolutionary Algorithms. In *Proceedings to Norwegian Conference on Computer Science*, 2001.

[77] Michael J Pugliese and Jonathan Cagan. Capturing a rebel: modeling the Harley-Davidson brand through a motorcycle shape grammar. *Research in Engineering Design*, 13(3):139–156, 2002.

[78] Francisco Regateiro, João Bento, and Joaquim Dias. Floor Plan Design using Block Algebra and Constraint Satisfaction. *Advanced Engineering Informatics*, 26(2):361–382, 2012.

[79] Aristides AG Requicha and Herbert B Voelcker. Constructive solid geometry. Technical Report Tech. Memo 25, Production Automation Project, Univ. of Rochester, 1977.

[80] Mark Riedl and Carlos León. Generating story analogues. In *AIIDE*, 2009.

[81] Elena Rishes, Stephanie M Lukin, David K Elson, and Marilyn A Walker. Generating different story tellings from semantic representations of narrative. In *International Conference on Interactive Digital Storytelling*, pages 192–204. Springer, 2013.

[82] Eleanor H Rosch. Natural categories. *Cognitive psychology*, 4(3):328–350, 1973.

[83] James Ryan, Ethan Seither, Michael Mateas, and Noah Wardrip-Fruin. Expressionist: An authoring tool for in-game text generation. In *International Conference on Interactive Digital Storytelling*, pages 221–233. Springer, 2016.

[84] Peter Simons. *Parts: A Study in Ontology*. Oxford University Press, 1987.

[85] Adam M Smith and Michael Mateas. Variations Forever: Flexibly Generating Rulesets from a Sculptable Design Space of Mini-games. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 273–280. IEEE, 2010.

[86] Adam M Smith and Michael Mateas. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011.

[87] Anthony J Smith and Joanna J Bryson. A Logical Approach to Building Dungeons: Answer Set Programming for Hierarchical Procedural Content Generation in Roguelike Games. In *Proceedings of the 50th Anniversary Convention of the AISB*, 2014.

[88] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 209–216. ACM, 2010.

[89] Thomas Smith, Julian Padget, and Andrew Vidler. Graph-based generation of action-adventure dungeon levels using Answer Set Programming. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, page 52. ACM, 2018.

[90] Robert Speer and Catherine Havasi. ConceptNet 5: A large semantic network for relational knowledge. In *The People's Web Meets NLP*, pages 161–176. Springer, 2013.

[91] Martin Stacey. Psychological challenges for the analysis of style. *AI EDAM*, 20(3):167–184, 2006.

[92] George Stiny. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3):343–351, 1980.

[93] George Stiny and William J Mitchell. The palladian grammar. *Environment and Planning B: Planning and Design*, 5(1):5–18, 1978.

[94] Adam Summerville. Expanding expressive range: Evaluation methodologies for procedural content generation. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE*, 2018.

[95] Brandon Tearse, Peter Mawhorter, Michael Mateas, and Noah Wardrip-Fruin. Skald: Minstrel reconstructed. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2):156–165, 2014.

[96] Romain Testuz, Yuliy Schwartzburg, and Mark Pauly. Automatic Generation of Constructable Brick Sculptures. In *Eurographics (Short Papers)*, pages 81–84, 2013.

[97] Mariët Theune, Nanda Slabbers, and Feikje Hielkema. The automatic generation of narratives. *LOT Occasional Series*, 7:131–146, 2007.

[98] Jukka Toivanen, Matti Järvisalo, Hannu Toivonen, et al. Harnessing constraint programming for poetry composition. In *The Fourth International Conference on Computational Creativity*. The University of Sydney, 2013.

[99] Scott R Turner. MINSTREL: A computer model of creativity and storytelling. 1994.

[100] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. *Instant architecture*, volume 22. ACM, 2003.

[101] Bernard Yannou and Ghassen Harmel. Use of Constraint Programming for Design. In *Advances in Design*, pages 145–157. Springer, 2006.

[102] Derek Yu. Spelunky (PC game), 2009.